VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SIMULACE KAPALIN NA GPU

GPU ACCELERATED FLUID SIMULATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. IGOR FRANK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ MILET**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Frank Igor, Bc.**
Program: Informační technologie
Obor: Počítačová grafika a multimédia
Název: **Simulace kapalin na GPU**
Fluid Simulation Using GPU
Kategorie: Počítačová grafika

Zadání:

1. Nastudujte vědecké publikace o simulaci kapalin. Nastudujte API Vulkan a akceleraci pomocí GPU.
2. Vyberte si vhodnou metodu simulace kapalin.
3. Implementujte vybranou metodu.
4. Navrhněte a implementujte rozšíření metody.
5. Proměřte a zhodnoťte výsledky.
6. Vytvořte demonstrační video.

Literatura:

- Dle doporučení vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2 a kostra aplikace.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Milet Tomáš, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 30. října 2020

Abstrakt

Tato práce se zabývá simulací kapalin, konkrétněji se zaměřuje na propojení částicové a mřížkové simulace modelující vypařování. Přistup k simulaci vypařování čerpá z článku Evaporation and Condensation of SPH-based Fluids autorů Hendrika Hochstettera a Andrease Kolba. Cílem celé práce však není jen tvorba simulace, ale zároveň studium různých metod používaných při simulaci kapalin.

Abstract

This thesis focuses on fluid simulation, particularly on coupling between particle based simulation and grid based simulation and thus modeling evaporation. Mentioned coupling is based on the article Evaporation and Condensation of SPH-based Fluids of authors Hendrik Hochstetter a Andreas Kolb. The goal of this thesis is not purely implementing ideas of the mentioned article, but also study of different methods used for fluid simulation.

Klíčová slova

simulace kapalin, Navier-Stokesovy rovnice, mřížková metoda simulace, SPH, vypařování, GPU, Vulkan

Keywords

fluid simulation, Navier-Stokes equations, grid-based simulation, SPH, evaporation, GPU, Vulkan

Citace

FRANK, Igor. *Simulace kapalin na GPU*. Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Milet

Simulace kapalin na GPU

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Igor Frank
19. května 2021

Poděkování

Chtěl bych poděkovat svému vedoucímu panu Ing. Tomáši Miletovi za rady při konzultaci a výběru tématu. Dále děkuji všem, kteří si nejen přečetli moji práci, ale také těm, kteří mi jakkoliv pomohli při tvorbě samotného programu a upozornili mě na vyskytující se nedostatky.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Nejprve je nutné poznamenat, že zadání práce bylo velice obecné. Na zadání „Simulace kapalin na GPU“ může být aplikováno mnoho řešení, a to nejen v rámci modelovaného fenoménu, ale i použitých technologií a metod. Lze například simulovat jednoduchou scénu padajícího objemu dané kapaliny. Další možností je například simulace interakce mezi dvěma druhy kapalin. Bylo by i možné vytvořit systém eroze, kdy by kapalina deformovala a ničila podloží a obecně pevná tělesa vyskytující se v systému. Další možností simulace je zaměřit se detailněji na konkrétní fyzikální vlastnosti kapalin, jako je například adheze či povrchové napětí. Zároveň je dostupných mnoho metod, jak k simulaci přistoupit. Lze využít mřížkové metody počítající hodnoty v předem definovaných místech, či redukovat paměťovou náročnost a využít řídkých mřížek. Možností je i využití celulárních automatů pro velmi jednoduché simulace, či využít například částicového přístupu pro mnohem složitější a fyzikálně přesnější simulace.

Tato práce se nejvíce zaměřovala na modelování fyzikálních vlastností kapalin. Přesněji řečeno se jedná o simulaci vypařování. V celém systému jsou jasně odlišitelné jednotlivé fáze a kapalina v závislosti na teplotě plynule přechází ze skupenství kapalného do plynného. V systému se zobrazuje jasná hladina s kapalinou a její výpary, pokud dochází k vypařování. V aplikaci je možné také celé vypařování vypnout a simulovat pouze tok kapaliny. Co se týče použitých metod, jsou v práci zastoupeny dva hlavní přístupy pro modelování toku kapalin či plynů. Pro simulaci kapaliny je využito částicového přístupu, přesněji metody Smoothed Particle Hydrodynamics představené autory Monaghan [15] a Lucy [27]. Pro simulaci plynného skupenství je pak využito mřížkové metody představené v roce 1999 autorem Jos Stamem [36]. Aby docházelo ke správnému přenosu informací mezi jednotlivými systémy, bylo nutné následně propojit obě metody mezi sebou.

Představovaný systém lze pak využít například při demonstraci daného jevu studentům a ukázání chování kapaliny v různých situacích. Nicméně hlavním cílem této práce je nejen vytvořit daný systém, ale především prozkoumat možnosti simulace kapalin. Primárním cílem tedy bylo prozkoumat různé metody modelování kapalin a tyto poznatky následně uplatnit při tvorbě samotné aplikace, čehož bylo docíleno zvolením článku [20] autorů Hendrika Hochstettera a Andrease Kolba z University of Siegen v Německu, zabývajícího se kondenzací a vypařováním a využívající dvě různé metody simulace.

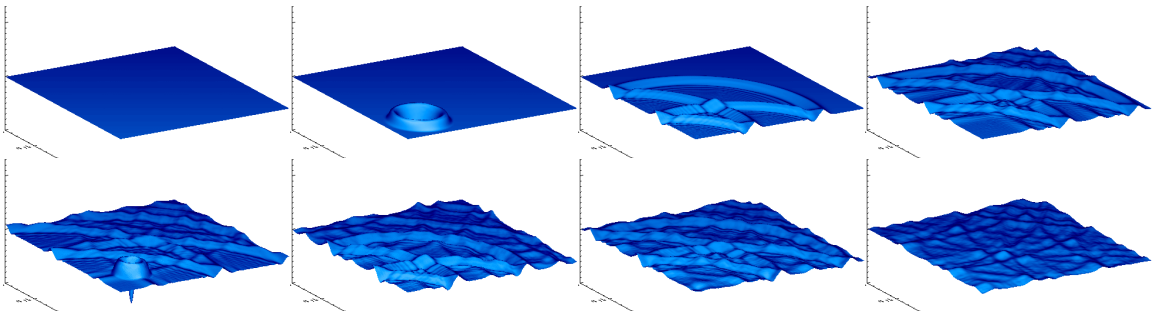
Kapitola 2

Teorie

Následující kapitola se věnuje vysvětlení základní teorie skrývající se za simulacemi a animacemi tekutin a plynů. Nejprve je zde obecně vysvětlen pojem simulace a animace kapalin. Dále jsou zde vysvětleny různé přístupy k simulaci, je zde vysvětlena mřížková neboli Eulerova metoda a částicový přístup neboli Lagrangeova metoda. V každé sekci jsou následně popsány některé významné algoritmy spadající pod jednotlivé metody.

2.1 Simulace kapalin

Nejprve je nutné odlišit pojmy simulace kapalin a animace kapalin. V obou případech jde především o chování kapaliny v určité situaci, nicméně animace se oproti simulaci zaměřuje především na vizuální stránku a méně na fyzikální přesnost. Jedná se tedy pouze o aproximaci vzorců popisujících chování kapalin například zanedbáním velké části objemu kapaliny a popisem pouze chování hladiny kapaliny (viz obrázek 2.1). Jedním z možných využití takových aproximací jsou případy, kde natolik nezáleží na přesnosti chování, jako spíše na vizuální stránce. Příkladem mohou být videohry, ve kterých je nutné mít vizuálně přívětivou kapalinu, ale zároveň vypočitatelnou v reálném čase s přihlédnutím na výpočet mnoha dalších jevů v rámci herního enginu. Avšak například při tvorbě animací a vizuálních efektů pro filmy, kde nejsme omezeni časem, lze využívat mnohem výpočetně náročnějších postupů. Tyto animace sice stále nemusí být fyzikálně dokonalé, jedná se však o mnohem propracovanější vyobrazení kapalin, než pouhé vlnění hladiny. [28]



Obrázek 2.1: **Shallow Water Equation.** Jedná se o poměrně jednoduchou metodu simulace hladiny vody, která však nedokáže zachytit veškeré vlastnosti chování kapalin. Obrázky zachycují postupné šíření vln při několika kapkách vody.
Zdroj: https://en.wikipedia.org/wiki/Shallow_water_equations

Zobrazení hladiny

Animace vlnění hladiny je jedna z nejjednodušších vizualizací kapaliny. Existuje samozřejmě více přístupů jak takovou animaci realizovat, mezi které patří například výškové mapy, vlnová funkce či rovnice mělké vody (Shallow Water Equation) zobrazené na obrázku 2.1. Ačkoliv zmíněné přístupy produkují v celku uspokojivé vlnění hladiny, existují v okolním světě běžné jevy, jako například lámající se vlny, které za pomocí jednoduchých vlnových funkcí a výškových map nelze simulovat. [28]

Obrázek 2.2: **Lámající se vlna.** Jev v reálném světě, který není realizovatelný za pomocí výškových map a vlnových funkcí.
Zdroj: https://en.wikipedia.org/wiki/Breaking_wave

Simulace kapalin se naopak snaží být oproti animacím co nejvíce fyzikálně přesné, čímž ale rapidně vzrůstá náročnost výpočtů. Existuje nespočet metod jak dosáhnout výsledku, nicméně čím přesnější a složitější scéna, tím delší výpočet daného scénáře. Doba výpočtu se tak může pohybovat v rámci minut, ale i hodin. Tyto metody jsou pak často založeny nad numerickými výpočty fyzikálních rovnic, kde nejpoužívanější rovnice jsou Navier-Stokesovy rovnice.

2.1.1 Navier-Stokesovy rovnice

Navier-Stokesovy rovnice jsou jedny z nejvyužívanějších rovnic pro výpočty chování kapalin. Základy pro popis dynamiky kapalin položil již v roce 1687 Sir Isaac Newton v článku Principa, kde bylo poprvé správně popsáno chování viskózních kapalin. Později Daniel Bernoulli a Leonhard Euler popsali chování neviskózního toku pomocí rovnic dnes známých jako Eulerovy neviskózní rovnice (Euler's inviscid equations). Až Claude-Louis Navier a Sir George Stokes na sobě nezávisle odvodili finální podobu rovnic ze zmiňovaných Eulerových a Newtonových rovnic. Tyto Navier-Stokesovy rovnice (rovnice 2.1 a 2.2) jsou nyní nejpoužívanější formou popisu chování kapalin a bylo na nich postaveno nespočet různých algoritmů. [8]

$$\rho\left(\frac{\partial}{\partial t}+\mathbf{u}\cdot\nabla\right)\mathbf{u}=-\nabla p+\mu\nabla\cdot(\nabla\mathbf{u})+f \tag{2.1}$$

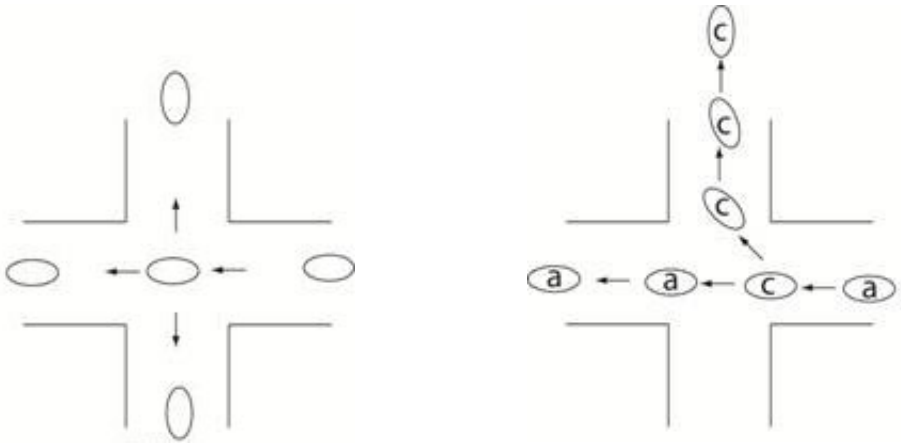
$$\nabla\cdot\mathbf{u}=0 \tag{2.2}$$

Tři základní vlastnosti viskózní kapaliny, která má neměnné teplo, jsou rychlost ($\mathbf{u}$), tlak (p) a hustota (ρ). V rovnici 2.1 pak μ označuje viskozitu kapaliny a f ostatní síly

působící na kapalinu, jako například gravitace. Tyto rovnice tudíž vyjadřují zákon o zachování hybnosti a zákon o zachování hmotnosti pro newtonské kapaliny. Newtonská kapalina je kapalina, kterou lze popsat lineárním Newtonovým modelem. Její viskozita je závislá především na tlaku a teplotě a z tohoto hlediska se jedná o takzvanou newtonskou viskozitu. U nenewtonské kapaliny pak popisujeme takzvanou zdánlivou viskozitu závislou na předchozí deformaci kapaliny a rychlosti vnitřního smyku kapaliny. Tyto kapaliny pak nelze popsat lineárním Newtonovým zákonem.[37]

Navier-Stokesovy rovnice jsou hojně využívány nejen pro simulaci a animaci kapalin, ale v celé řadě dalších vědních oborů. Využití můžeme nalézt při vytváření modelů pro předpověď počasí, pro studování toku vzduchu při výrobě letadel nebo například při analýze šíření znečištění.

Existují dva úhly pohledu na organizaci a řešení nejen těchto rovnic, ale zároveň obecně na řešení dynamiky kapalin. Tyto metody se pak liší především v bodech pozorování. V Eulerově metodě dochází k výpočtu požadovaných hodnot v přesně daných bodech, tedy v určité diskrétní mřížce. U Lagrangeovy metody dochází k výpočtu v bodech sledované masy, tedy v částicích sledované kapaliny. Níže jsou obě metody detailněji popsány, včetně několika algoritmů, které pod dané metody spadají.



(a) **Eulerův přístup** (b) **Lagrangeův přístup**

Obrázek 2.3: **Rozdíl mezi Eulerovým a Lagrangeovým přístupem.** Příklad dvou přístupů nad křižovatkou s auty. Eulerův přístup (2.3a) sleduje křižovatku v předem daných místech (ramena a střed křižovatky). Naopak Lagrangeův přístup(2.3b) sleduje konkrétní vozidla (a,c), jak projíždí křižovatkou.
Zdroj:
http://abe-research.illinois.edu/faculty/dickc/Engineering/ELdescrip2a.htm

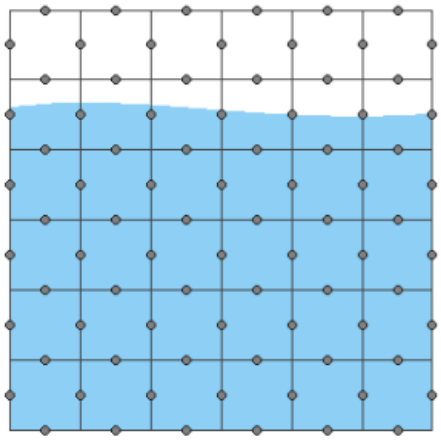
2.2 Eulerova metoda toku

Jak bylo popsáno výše, při využití Eulerovy metody jsou vlastnosti kapaliny počítány v pevně definovaných bodech diskrétní mřížky. Ačkoliv některé vlastnosti kapaliny popisuje Eulerova metoda mnohem přesněji, největší nevýhodou je samotná mřížka. Jeden z prvních problémů je hrubost mřížky. Při pohledu na obrázek 2.4 je patrné, že reálná hladina je lehce

zvlněná, ale z důvodu velké hrubosti mřížky a tedy hrubějšímu vzorkování, by lehké zvlnění bylo zanedbáno a hladina by byla rovná.

Dalším problémem je paměťová náročnost. Je-li využito jemnější mřížky, z důvodu odstranění problémů souvisejících s hrubou mřížkou, je nutno počítat se zvyšující se paměťovou náročností. Při výpočtech v trojrozměrném prostoru a s dvojnásobným zpřesněním v každé ose, dojde k osminásobně zvýšenému počtu buněk v paměti. Tento problém však naštěstí lze již řešit pomocí různých struktur jako jsou například řídké mřížky, u kterých dochází k zvýšené jemnosti pouze v potřebných oblastech.

Posledním problémem je samotná uzavřenost mřížky, která brání k plně dynamické simulaci. Kapalina je tedy uzavřena do „nerozbitné nádoby“ a jakýkoliv pokus o její tok mimo mřížku je nemožný. I na tento problém však existuje řešení, a to v podobě dynamických mřížek, které se v případě nutnosti dokáží rozšiřovat v prostoru a poskytují tak možnosti pro rozsáhlejší simulační prostor. [22]



Obrázek 2.4: **Eulerova mřížka.** Kapalina uzavřená ve 2D mřížce. Rychlost a ostatní vlastnosti kapaliny jsou reprezentovány pouze ve vyznačených tečkách. Z důvodu hrubosti mřížky, pak může docházet k zanedbání některých detailů.
Zdroj: Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics [22]

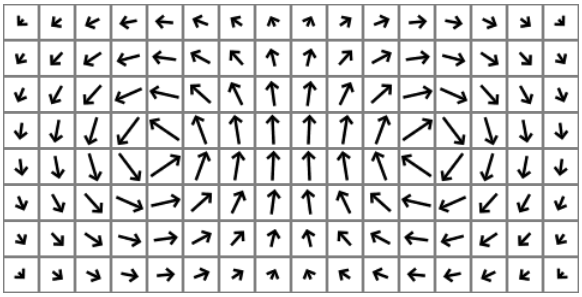
2.2.1 Mřížková metoda

Mřížková (Grid) metoda je Eulerovská metoda využívající pro pohyb kapaliny několika polí vektorů. Základem mřížkové metody je pole vektorů rychlostí pro každý zkoumaný bod v mřížce. Toto pole pak představuje pohyb tekutiny v celém zkoumaném prostoru. Pole rychlostí lze vyjádřit rovnicí 2.3

$$\vec{u}(x, y) = (u_x, u_y) \tag{2.3}$$

Dalším základním stavebním kamenem algoritmu je pak advekce, neboli přesun vlastností z jednoho místa na jiné v důsledku pohybu kapaliny. Je-li uvažováno, že tok kapaliny přenáší například určitou koncentraci částic (kouř, barvivo, písek), pak existují dvě možnosti, jak posunout dané hodnoty v čase a prostoru. První možností je dopředný posun v čase vyjádřen rovnicí 2.4. V závislosti na pozici $\mathbf{r}$ v prostoru je zvolena odpovídající rychlost $\mathbf{u}$ a daná hodnota A je posunuta v prostoru.

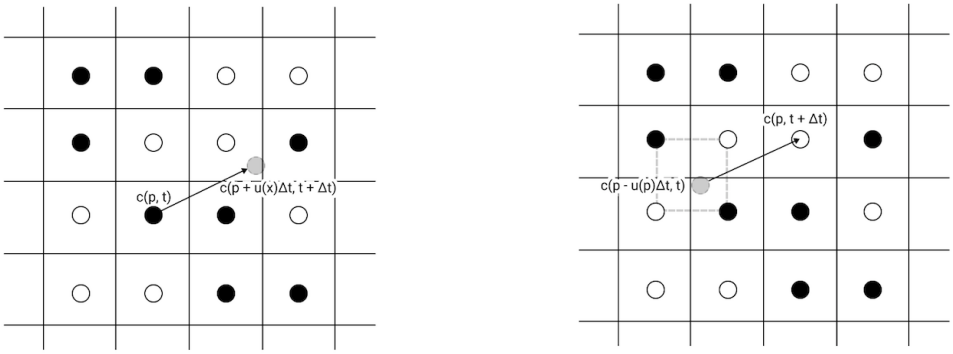
$$A(\mathbf{r} + \mathbf{u}\Delta t, t + \Delta t) = A(\mathbf{r}, t) \tag{2.4}$$



Obrázek 2.5: **Pole rychlostí.** Pomocí šipek je vizuálně znázorněn směr a velikost rychlosti.
Zdroj: https://www.karlsims.com/fluid-flow.html

Druhým možným přístupem pro přesun hodnot je tzv. backtracking. V tomto přístupu není hodnota posunuta z jedné pozice na druhou. Vektor rychlosti se naopak invertuje a do nynější pozice se přesouvá pozice předchozí. Následující rovnice 2.5 popisuje právě tuto metodu. [7]

$$A(\mathbf{r}, t + \Delta t) = A(\mathbf{r} - \mathbf{u}\Delta t, t) \tag{2.5}$$

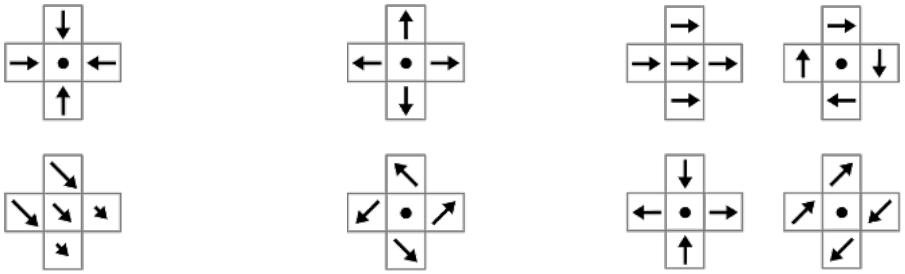


(a) **Dopředná advekce**

(b) **Zpětná advekce**

Obrázek 2.6: **Příklad dvou přístupů advekce.** Na obrázku 2.6a lze vidět dopředný posun hodnot, kdy je například hustota posouvána vpřed, ve směru vektoru rychlosti. Na obrázku 2.6b je následně znázorněn zpětný posun, kdy dochází k otočení vektoru rychlosti a pomocí interpolace se zjišťuje jaká hodnota „doputovala“ na současnou pozici.
Zdroj: Fluid Simulation (with WebGL demo) [7]

Stejně jako dochází k přesunu určité hodnoty v prostoru, lze přesouvat i pole rychlostí a měnit jej tak v čase. V tomto případě však můžou nastat problémy s nestlačitelností a zákonem zachování hmotnosti. Je nutné zaručit, že divergence pole rychlosti je všude nulová. Divergence zjednodušeně říká, zda v určitém bodě daná vlastnost roste, či klesá. Pokud je vyžadováno docílit nulové divergence pole rychlosti, jedná se o docílení stavu, kde v žádném bodě neklesá ani neroste hustota kapaliny. [7]



(a) **Záporná divergence** (b) **Kladná divergence** (c) **Nulová divergence**

Obrázek 2.7: **Příklady divergence.** Příklady různých divergencí v bodě v závislosti na velikosti a směru vektorového pole v okolí. Na obrázcích 2.7a a 2.7b si lze všimnout, že v prostřední buňce dochází k přílišnému hromadění či úbytku hodnot.
Zdroj: https://www.karlsims.com/fluid-flow.html

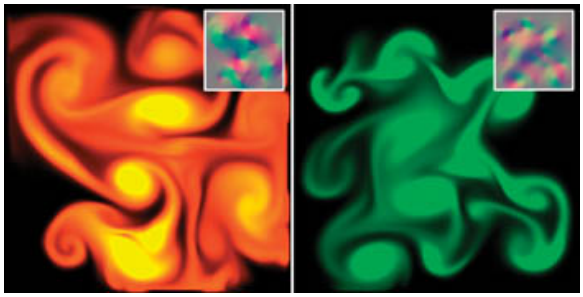
Při výpočtu pole s nulovou divergencí lze využit takzvaný Helmholtz-Hodgeův rozkladový teorém 2.6,

$$\mathbf{u} = \mathbf{w} - \nabla p \tag{2.6}$$

kde $\mathbf{u}$ je zmíněné hledané pole s nulovou divergencí, $\mathbf{w}$ je pole rychlostí s nenulovou divergencí a ∇p je gradient tlaku. Tato rovnice tedy říká, že pole rychlostí s nenulovou divergencí může být opraveno pomocí gradientu tlaku. Pro výpočet tlaku lze pak odvodit ze stejného teorému 2.6 následující rovnici 2.7. [7]

$$p_{x,y}^{(k+1)} = \frac{p_{x+1,y}^{(k)} + p_{x-1,y}^{(k)} + p_{x,y+1}^{(k)} + p_{x,y-1}^{(k)} + \alpha b_{x,y}}{\beta} \tag{2.7}$$

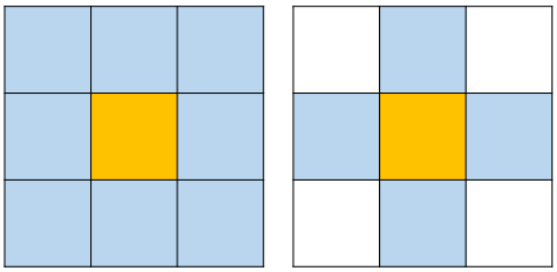
Přičemž pro neviskózní kapaliny platí, že v rovnici 2.7 mají konstanty následující hodnoty $\alpha = -(velikost\_bunky)^2$, $\beta = 4$ a $b = \nabla \cdot \mathbf{w}$. Tato rovnice je pak řešena například Jacobiho iterativní metodou, kde počáteční odhad je nulový tlak ve všech bodech. Po dostatečném počtu iterací je výsledkem hodnota tlaku ve všech bodech a po výpočtu gradientu a aplikování ve vzorci 2.6 je získáno pole rychlostí s nulovou divergencí. Pomocí tohoto pole pak lze posouvat výše zmíněné hodnoty jako je barva, koncentrace částic a jiné. [6]



Obrázek 2.8: **Mřížková metoda.** Vizualizace barviva v proudící vodě. Výsledek implementované metody Eulerovy mřížky z knihy GPU Gems.
Zdroj: Chapter 38. Fast Fluid Dynamics Simulation on the GPU [6]

2.2.2 Celulární automaty

Pro simulaci toku kapalin lze využít i celulárních automatů. Celulární automat se skládá z několika důležitých částí, stavového prostoru rozděleného na diskrétní buňky, přechodové funkce a množiny stavů, které mohou jednotlivé buňky nabývat. Celulární automaty pracují na principu n-okolí, kde n závisí na dimenzi zkoumaného prostoru a typu okolí. V jednom časovém kroku se pak aplikuje na všechny buňky přechodová funkce, která vyhodnotí stavy okolních buněk a podle výsledku nastaví vlastní stav.



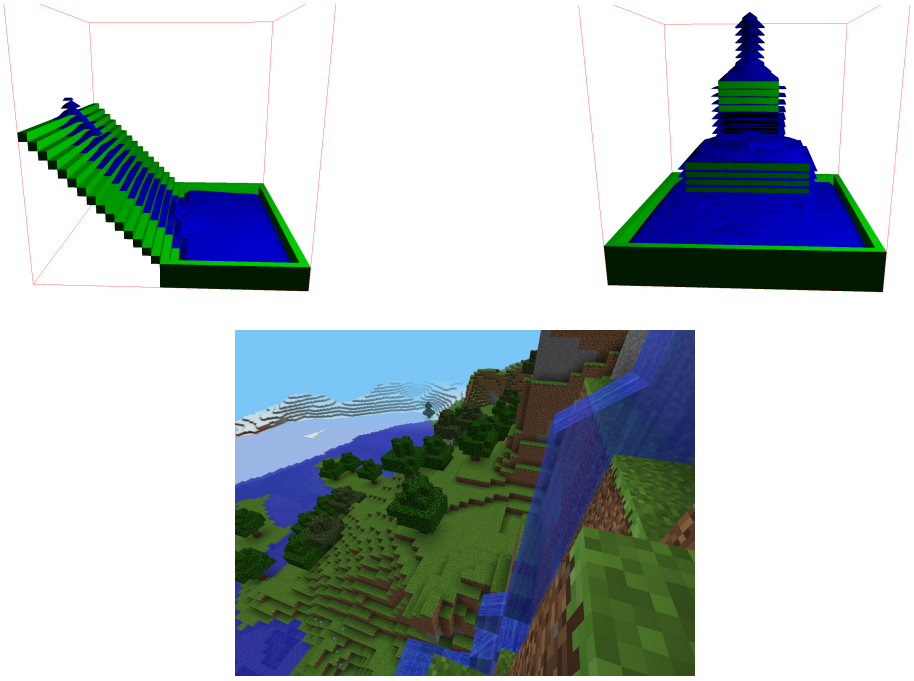
Obrázek 2.9: **Typy okolí buňky.** Na obrázku vlevo se nachází Moorovo osmi okolí, vpravo poté Von-Neumanovo čtyř okolí. Zobrazeny jsou dvourozměrné verze okolí, převod do třetího rozměru je triviální.
Zdroj:
http://complextin.blogspot.com/2016/06/2d-cellular-automata-three-state.html

Pomocí popsaného principu a využití několika jednoduchých pravidel pak lze vytvořit velice jednoduchý simulátor toku vody. Nejdříve se simuluje působení gravitace a automat se pokusí co nejvíce vody přesunout o buňku níže. Pokud je buňka plná nebo se jedná o buňku, do které nemůže téci voda, zbylá kapalina je distribuována do okolních horizontálních buněk. Jedná se o velice jednoduchý algoritmus, který však zanedbává mnoho vlastností kapalin. Prvním problémem je situace při spojených nádobách, kdy pomocí tohoto algoritmu nedojde k vyrovnání hladin a je nutné aplikovat další procesy pro vyrovnávání hladin. Dalším problémem je odrazivost a rychlost toku kapalin. Kapalina se bude vždy pohybovat stejnou rychlostí a představíme-li si situaci, kdy „naráží“ do stěny, pak nedojde k roztříštění a případnému stoupání kapaliny vzhůru. Dalším problémem mohou být vlastnosti kapaliny, jako je například viskozita, která se zanedbává, případně může být simulována pouhým koeficientem ovlivňujícím rychlost toku. [28]

2.3 Lagrangeova metoda toku

Základem Lagrangeovy metody toku není sledování vymezeného prostoru, kde se může kapalina pohybovat, jako je tomu u Eulerovy metody, ale sledovaní kapaliny samotné. Lagrangeova metoda se tak soustředí na celkovou kapalinu, kterou dělí na samostatné části. Lze tedy říci, že se jedná o částicovou metodu, kde každá taková částice kapaliny nese specifické parametry jako hustotu, tlak, hmotnost a jiné. Pomocí těchto parametrů pak ovlivňuje ostatní částice v okolí a tím i celou masu kapaliny.

Jak z popisu vyplývá, tyto metody nejsou v prostoru omezeny žádnou mřížkou či jinou strukturou, limitující prostor dané simulace. Přestože nevzniká vysoká paměťová náročnost z nutnosti mít mřížku, vyvstává paměťová náročnost v podobě počtu částic. Pro dostatečně



Obrázek 2.10: **Příklady celulárních simulací.** Na horních obrázcích je simulace vytvořená během studia společně s Bc. Petr Flajšingr. Na spodním obrázku je ukázána simulace vody ve známé hře Minecraft. Oba příklady představují poměrně jednouduchý simulátor vody za použití celulárních automatů.
Zdroj: https://github.com/Nirvanios/Water_simulation-cellular_automaton
Zdroj: https://mcpedl.com/ayay/

přesnou a jemnou simulaci je totiž nutné mít vysoký počet částic. Počty částic se mohou pohybovat v řádech od desítek tisíc až po jednotky či desítky miliónů. Při předpokladu vysokého počtu částic a skutečnosti, že každá částice o sobě musí nést mnoho informací jako je rychlost, pozice, hmotnost a další, je nutné počítat s paměťovými nároky na jiném místě.

2.3.1 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (dále jen SPH) je dnes jednou z nejpoužívanějších metod částicové simulace kapalin. Tato metoda byla představena autory Gingold a Monaghan [15] a Lucy [27], původně pro řešení problému v teoretické astrofyzice. Postupem času se z ní ale stala velice populární metoda, která se dočkala nespočtu rozšíření a modifikací a začala se používat nejen v astrofyzice, ale také v balistice, vulkanologii a především obecně v oblastech simulace tekutin a jiných spojitých mas.

SPH je Lagrangeova metoda, základem jsou tedy částice. Každá částice nese několik konkrétních hodnot, jako je hmotnost, pozice a vektor rychlosti a navíc ještě několik dalších hodnot, vztahujících se k danému problému, jako je hustota hmotnosti, tlak nebo například teplota. Jedná se o integrační metodu, kde je výpočet konkrétního atributu $A(\mathbf{r})$ částice I nad prostorem Ω definován pomocí rovnice 2.8.

$$A_I(\mathbf{r}) = \int_\Omega A(\mathbf{r}')W(\mathbf{r} - \mathbf{r}', h)\mathbf{dr}' \tag{2.8}$$

V této rovnici $\mathbf{r}$ značí pozici v prostoru, W je vyhlazovací jádro (smoothing kernel) a h je šířka jádra. Šířka jádra, nazývaná také vyhlazovací poloměr (smoothing radius), pak ovlivňuje kvalitu a stabilitu simulace. Z podmínek platících pro jádro popsaných níže, lze následně odvodit, že nová hodnota daného atributu je určitým průměrem hodnot okolních částic. Numerický výpočet je následně zobrazen rovnicí 2.9,

$$A_S(\mathbf{r}) = \sum_j A_j V_j W(\mathbf{r} - \mathbf{r_j}, h) \tag{2.9}$$

$$V = \frac{m}{\rho} \tag{2.10}$$

kde dochází k aproximaci integrálu pomocí sumy přes všechny částice j a V je objem v prostoru, který částice zaujímá, přičemž je vypočítaný pomocí známé rovnice 2.10, kde m je hmotnost částice a ρ je hustota hmotnosti částice. [22]

Vyhlazovací jádro

Vyhlazovací jádro je jednou z nejdůležitějších součástí celého SPH algoritmu a výběr správného jádra má tedy značný vliv na kvalitu a stabilitu simulace. Vhodné jádro musí splňovat dvě důležité podmínky.

$$\int_\Omega W(\mathbf{r} - \mathbf{r}', h)\mathbf{dr}' = 1 \tag{2.11}$$

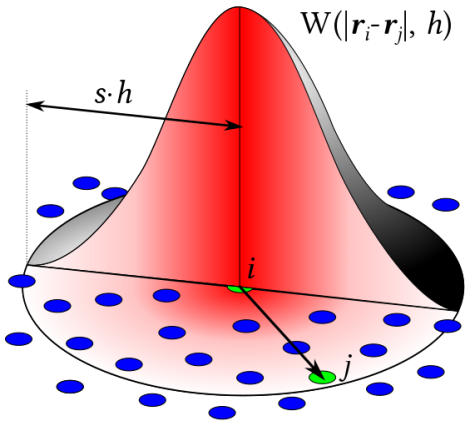
$$\lim_{h \to 0} W(\mathbf{r} - \mathbf{r}', h) = \delta(\mathbf{r} - \mathbf{r}') \tag{2.12}$$

$$\delta(x) = \begin{cases} \infty & ||\mathbf{r}|| = 0 \\ 0 & \text{jinak} \end{cases} \tag{2.13}$$

První podmínka 2.11 udává, že jádro musí být normalizované, a nemůže tedy nikterak ovlivnit hodnoty, s nimiž pracuje. Druhá podmínka 2.12, kde δ značí Diracovu delta funkci 2.13, pak limituje počet interpolantů. Značí, že výsledek je ovlivněn pouze několika body v okolí a body příliš vzdálené v prostoru nikterak neovlivňují výsledek. V původním článku autoři pro výpočet použili jednodimenzionální Gaussovské jádro (rovnice 2.14). Jako další možnost je použít B-Splajn, kvintický splajn [25] nebo například polynomiální kernel šestého stupně (rovnice 2.15) [30].

$$W_{Gauss}(x, h) = \left(\frac{1}{h\pi}\right)^{\frac{3}{2}} \exp\left(-\frac{x^2}{h^2}\right) \tag{2.14}$$

$$W_{Poly6}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - ||\mathbf{r}||^2)^3 & 0 \leq ||\mathbf{r}|| \leq h \\ 0 & \text{jinak} \end{cases} \tag{2.15}$$



Obrázek 2.11: **Vizualizace vyhlazovacího jádra.** Zelený bod i je částice, pro kterou právě probíhá výpočet. Bod j je jeden z bodů ovlivňující počítaný atribut částice i. Parametr h následně udává poloměr jádra, který určuje oblast s částicemi přispívajícími k výpočtu. Samotné jádro je znázorněno trojrozměrnou Gaussovskou funkcí, která udává, jak velký význam daná částice má pro počítaný atribut. Vzdálenější částice nejsou tak významné v rámci výpočtu jako částice blíže k částici i.
Zdroj: https://en.wikipedia.org/wiki/Smoothed-particle_hydrodynamics

Hustota hmotnosti

První veličinou, kterou je nutné vypočítat, je hustota hmotnosti pro každou konkrétní částici. Algoritmus SPH předpokládá, že hmotnost všech částic je během algoritmu neměnná a zároveň dopředu známá hodnota. Po kombinaci rovnic 2.9 a 2.10 a dosazení hustoty ρ jako počítaného atributu, vznikne následující rovnice 2.16 pro výpočet hustoty.

$$\begin{aligned} \rho_i &= \sum_i \rho_j V_j W(\mathbf{r_i} - \mathbf{r_j}, h) \\ &= \sum_i m_j W(\mathbf{r_i} - \mathbf{r_j}, h) \end{aligned} \tag{2.16}$$

Tlak

Tlak a síly působící díky tlaku jsou nesmírně důležité z důvodu zachování nestlačitelnosti kapaliny. Dostane-li se větší shluk částic do jednoho místa, dojde k lokálnímu zvýšení hustoty a tedy i tlaku a v důsledku toho je nutné aplikovat takové síly, aby se shluk rozprostřel na větším prostoru. Tato síla je pak v Navier-Stokesových rovnicích 2.1 reprezentována termem $-\nabla p$. Pro výpočet tlakové síly působící na částici, lze opět aplikovat rovnici 2.9, na ni známý vzorec 2.10 a dosadit právě zmíněný term z Navier-Stokesových rovnic.

$$f_i^{tlak} = -\nabla p_i = \sum_i p_j \frac{m_j}{\rho_j} \nabla W(\mathbf{r_i} - \mathbf{r_j}, h) \tag{2.17}$$

Výsledná rovnice 2.17 by však neprodukovala správné výsledky. Je-li uvažována simulace se dvěma částicemi, lze přijít na jeden zásadní problém, a to, že pomocí této rovnice není možné obdržet symetrické síly. Každá částice má lehce odlišný tlak, a proto tedy budou působící síly asymetrické. Existuje více způsobů jak symetrizovat tyto síly v kontextu SPH. Nejjednodušším příkladem je následující rovnice 2.18. [22] [29]

$$f_i^{tlak} = -\sum_i \frac{p_i + p_j}{2} \frac{m_j}{\rho_j} \nabla W(\mathbf{r_i} - \mathbf{r_j}, h) \tag{2.18}$$

Nyní, po získání symetrické tlakové síly, je poslední neznámou hodnota parametru p, označujícího tlak v místě částice. Tuto hodnotu tlaku lze obdržet ze stavové rovnice ideálního plynu 2.19,

$$p = k\rho \tag{2.19}$$

kde p je hledaný tlak, k je plynová konstanta závislá na teplotě a ρ je hustota vypočítaná pomocí rovnice 2.16. Pro dosažení lepších výsledků lze také využít následující rovnice 2.20,

$$p_i = k(\rho_i - \rho_0) \tag{2.20}$$

kde ρ_0 označuje klidovou hustotu kapaliny. Jelikož síla závisí na gradientu pole tlaku, tento posun neovlivní výsledek sil, nicméně může učinit simulaci o něco stabilnější.

Posledním problémem je výše popsané vyhlazovací jádro 2.15. Zmíněné jádro má totiž snižovat velikosti odpuzujících sil, čím blíže částice jsou. Nastane-li tedy vysoký shluk částic, jádro není schopné produkovat dostatečné síly pro rovnoměrné rozprostření částic. Proto bylo navrženo [13] lepší jádro 2.21, schopné dané částice od sebe odtrhnout.

$$W = (\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - \mathbf{r})^3 & 0 \leq ||\mathbf{r}|| \leq h \\ 0 & \text{jinak} \end{cases} \tag{2.21}$$

Viskozita

Při aplikaci dosavadních rovnic a postoupení k integraci (viz níže), je výsledkem simulace neviskózní kapaliny. V reálném světě má však za běžných podmínek každá kapalina určitou viskozitu. Viskozitu lze chápat jako působení sil bránící kapalině ve změně tvaru z důvodu vnitřního tření. Pro zavedení rovnice na výpočet síly působící díky viskozitě jsou použity opět Navier-Stokesovy rovnice, tentokrát term pro viskozitu $\mu\nabla^2\mathbf{u}(\mathbf{r_i})$ a opět obecnou rovnici pro SPH 2.9 společně s rovnici pro objem 2.10.

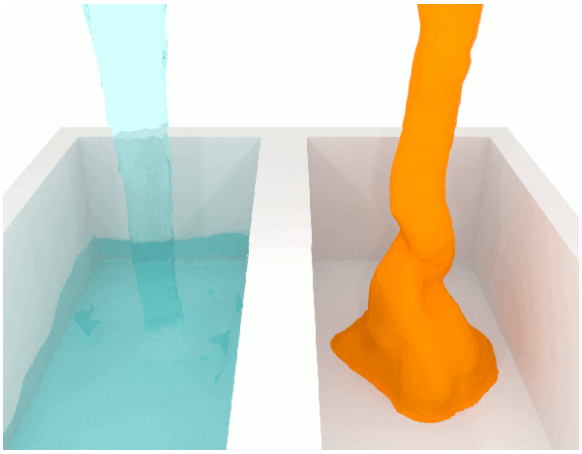
$$f_i^{viskozita} = \mu\nabla^2\mathbf{u}(\mathbf{r_i}) = \mu \sum_i \mathbf{u_j} \frac{m_j}{\rho_j} \nabla^2 W(\mathbf{r_i} - \mathbf{r_j}, h) \tag{2.22}$$

Výsledná rovnice 2.22 má však stejný problém, jako rovnice u tlakových sil, a to že jsou síly opět asymetrické. Správného výsledku tedy lze dosáhnout následující rovnicí 2.23. [30]

$$f_i^{viskozita} = \mu \sum_i m_j \frac{\mathbf{u_i} - \mathbf{u_j}}{\rho_j} \nabla^2 W(\mathbf{r_i} - \mathbf{r_j}, h) \tag{2.23}$$

Viskozita je vlastnost tekutiny, která do celého systému ze svého principu nepřináší energii, a tedy v celé tekutině nikterak nevzrůstá rychlost. Avšak jak polynomiální jádro šestého řádu 2.15, tak jádro použité pro výpočet tlaku 2.21, může do systému takové zvýšení energie zanést. Je nutné tedy najít takové jádro, které pouze snižuje velikost sil v závislosti na velikosti viskozity. Následující jádro 2.24 tuto podmínku splňuje. [30]

$$W = (\mathbf{r}, h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{||\mathbf{r}||^3}{2h^3} + \frac{||\mathbf{r}||^2}{h^2} + \frac{h}{2||\mathbf{r}||} & 0 \leq ||\mathbf{r}|| \leq h \\ 0 & \text{jinak} \end{cases} \tag{2.24}$$



Obrázek 2.12: **Viskozita kapalin.** Případy různé viskozity kapalin. Kapalina vpravo představuje viskóznější kapalinu a mnohem více se brání toku, přičemž se snaží si zachovat svůj tvar. Naopak kapalina vlevo je mnohem méně viskózní, a proto snadněji vyplňuje daný prostor.
Zdroj: https://en.wikipedia.org/wiki/Fluid_animation

Povrchové napětí

Povrchové napětí je síla působící na vnější povrch kapaliny, přesněji řečeno se jedná o nerovnováhu sil. Každá molekula kapaliny vytváří síly přitahující ostatní molekuly v kapalině, jež jsou poblíž. Uvnitř kapaliny jsou síly vyrovnané, ale jak znázorňuje obr. 2.13, molekuly na okraji kapaliny jsou pouze přitahovány do pomyslného středu kapaliny. Povrchové napětí sice není přítomno v Navier-Stokesových rovnicích, nicméně se jedná o velice důležitou součást chování kapalin. Povrchové napětí totiž zapříčiňuje minimalizaci povrchu kapaliny, kterou lze v přírodě pozorovat. [22]

$$f_i^{povrchové\ napětí} = \sigma \nabla^2 c_i \frac{\mathbf{n}_i}{||\mathbf{n}_i||} \tag{2.25}$$

Pomocí výše zmíněné rovnice 2.25 pak lze vypočítat sílu povrchového napětí. V rovnici σ představuje koeficient povrchového napětí, specifický pro danou kapalinu, c je tzv. barevné pole a $\mathbf{n}$ je normála povrchu kapaliny směřující dovnitř. Barevné pole c je definováno podmínkou 2.26.

$$c_i = \begin{cases} 1 & \text{na pozici částice } i \\ 0 & \text{jinde} \end{cases} \tag{2.26}$$

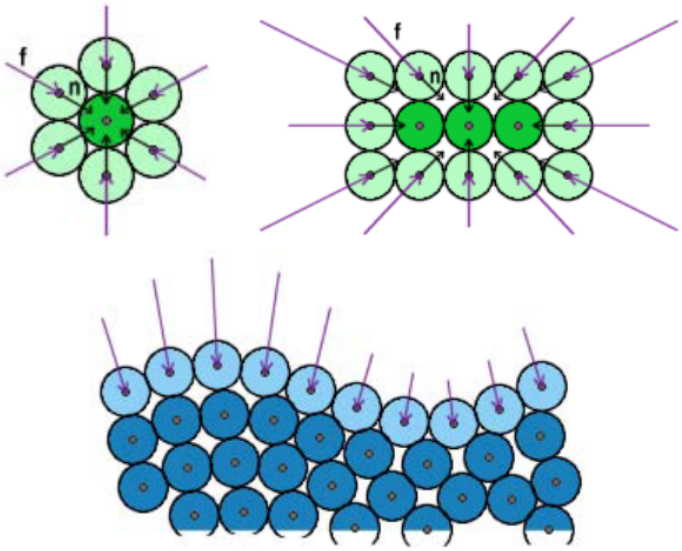
S pomocí rovnice 2.9 lze pak pole c_i vypočítat následující rovnicí 2.27.

$$\begin{aligned} c_i &= \sum_j c_j V_j \nabla^2 W(\mathbf{r_i} - \mathbf{r_j}, h) \\ &= \sum_j V_j \nabla^2 W(\mathbf{r_i} - \mathbf{r_j}, h) \end{aligned} \tag{2.27}$$

Z gradientu barevného pole pak lze získat normálu povrchu za pomocí následující rovnice 2.28 $\mathbf{n}_i$.

$$\mathbf{n}_i = \nabla c_i \tag{2.28}$$

Podmínka vyhodnocení je omezena členem $\frac{\mathbf{n}_i}{||\mathbf{n}_i||}$, kde $||\mathbf{n}_i||$ nabývá nenulových hodnot pouze blízko okraje kapaliny. Pokud je však $||\mathbf{n}_i||$ příliš malé, způsobuje numerickou nestabilitu řešení, a proto se síla povrchového napětí počítá jen v případě, kdy $||\mathbf{n}_i||$ překročí jistý práh.



Obrázek 2.13: **Povrchové napětí.** Různé případy povrchového napětí v kapalině. U zelené kapaliny jsou částice k sobě přitahovány tak, že jsou síly navzájem vyrušeny. U modré kapaliny jsou světle modré částice přitahovány k tmavě modrým a dochází tak působení sil, které vyvolává vlastnost zachování si nejmenšího povrchu kapaliny.
Zdroj: Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics [22]

Síly

Síly působící na kapalinu lze rozdělit do dvou kategorií, vnitřní a vnější síly. Pro výpočet celkové síly působící na jednu částici v systému pak stačí vnější a vnitřní síly sečíst. Vnitřní a vnější síly jsou definovány rovnicemi 2.29 a 2.30 následovně, [22]

$$f^{vnitřní} = f^{tlak} + f^{viskozita} \tag{2.29}$$

$$f^{vnější} = f^{povrchové\ napětí} + f^{gravitace} \tag{2.30}$$

kde je síla působící z důvodu gravitace vyjádřena rovnicí 2.31, přičemž g značí gravitační zrychlení.

$$f^{gravitace} = \rho g \tag{2.31}$$

Posledním důležitým krokem je výpočet zrychlení částice 2.32 z důvodu působení sil.

$$a_i = \frac{f^{vnější} + f^{vnitřní}}{\rho} \tag{2.32}$$

Integrace

Nyní představené vzorce jsou vše potřebné pro základní SPH simulaci, nicméně pouhým výpočtem výše popsaných vlastností hodnot nelze dostat tekoucí kapalinu. Je nutné každou částici vždy posunout v čase a prostoru na základě předchozí pozice, rychlosti a působících sil. Existuje několik přístupů jak numericky řešit posun částic v čase a prostoru.

Mezi nejznámější a nejjednodušší metodu numerické integrace patří Eulerova metoda. V její nejjednodušší formě se pozice i rychlost (rovnice 2.33 a 2.34) aktualizují zároveň na základě vypočtených hodnot.

$$\mathbf{u}_{t+\Delta t} = \mathbf{u}_t + \Delta t \mathbf{a}_t \tag{2.33}$$

$$\mathbf{r}_{t+\Delta t} = \mathbf{r}_t + \Delta t \mathbf{u}_t \tag{2.34}$$

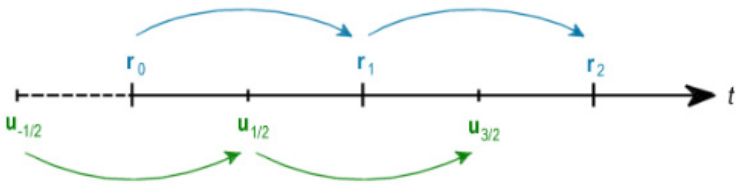
Existuje lehké zlepšení metody, ve kterém se následující pozice počítá z již vypočítané rychlosti. Pozice tedy závisí na výpočtu rychlosti. Při integraci je následně rovnice 2.34 nahrazena rovnicí 2.35. [22]

$$\mathbf{r}_{t+\Delta t} = \mathbf{r}_t + \Delta t \mathbf{u}_{t+\Delta t} \tag{2.35}$$

Dalším vylepšením může být integrační metoda Leap-Frog. Tato metoda vychází z Eulerovy metody, nicméně zavádí posunutí výpočtu rychlosti v čase. Výpočet rychlosti je posunut o $\frac{\Delta t}{2}$, čímž dochází ke „střídání“ výpočtu rychlosti a pozice (viz. obrázek 2.14). Vzorce pro výpočet rychlosti 2.36 a pozice 2.37 jsou následující.

$$\mathbf{u}_{t+\frac{\Delta t}{2}} = \mathbf{u}_{t-\frac{\Delta t}{2}} t \mathbf{a}_t \tag{2.36}$$

$$\mathbf{r}_{t+\Delta t} = \mathbf{r}_t + \Delta t \mathbf{u}_{t+\frac{\Delta t}{2}} \tag{2.37}$$



Obrázek 2.14: **Leap-Frog integrace.** Zobrazení vzájemného posunutí integrace rychlosti a pozice. Pozice **r** je počítána v čase t, zároveň s ní je počítána rychlost **u** pro čas $t + \frac{1}{2}$ vycházející z rychlosti **u** v čase $t - \frac{1}{2}$.
Zdroj: Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics [22]

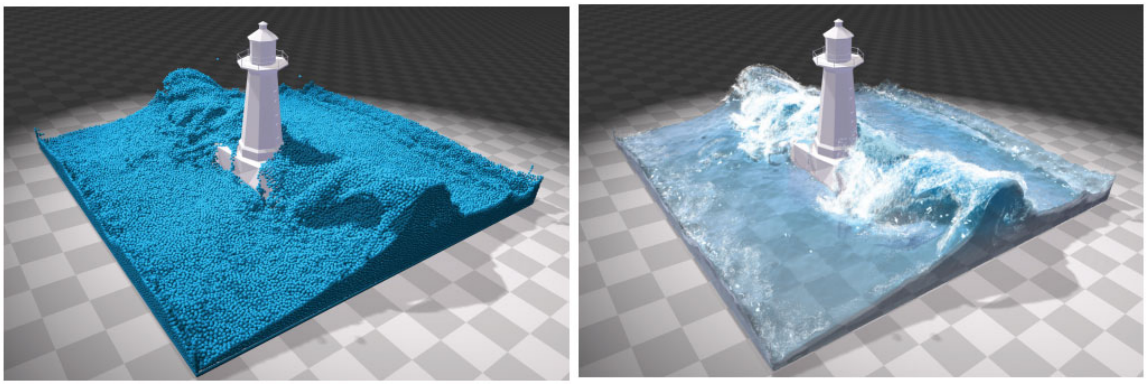
Algoritmus simulace

Algoritmus 1: Krok SPH Simulace

foreach *particle* **do**
- vypočti hustotu (viz. 2.16)
- vypočti tlak (viz. 2.20)

end foreach
foreach *particle* **do**
- vypočti tlakovou sílu f_i^{tlak} (viz. 2.18)
- vypočti viskózní sílu $f_i^{viskozita}$ (viz. 2.23)
- vypočti vlastní hodnotu barevného pole c_i (viz. 2.27)
- vypočti normálu povrchu kapaliny $\mathbf{n}_i$ (viz. 2.28)
- vypočti sílu povrchového napětí $f_i^{povrchové\ napětí}$ (viz. 2.25)
- $f^{vnitrni} = f_i^{tlak} + f_i^{viskozita}$
- $f^{vnitrni} = f_i^{povrchové\ napětí} + f_i^{gravitace}$
- $f = f^{vnější} + f^{vnitřní}$ (viz. 2.30 a 2.29)
- vypočti zrychlení a_i (viz. 2.32)
- integruj a posuň částici v čase (viz. 2.37 a (viz. 2.37)

end foreach



Obrázek 2.15: **Příklad SPH simulace.** Na levém obrázku je simulace zobrazená pomocí jednotlivých částic. Na pravém obrázku je ukázána vykreslená hladina kapaliny.
Zdroj: SPH Fluids in Computer Graphics [21]

Kapitola 3

Návrh řešení

Důležitou součástí procesu tvorby simulátoru je vytvořit si návrh celého systému. Je důležité zvolit vhodný jazyk pro implementaci, vhodné grafické/výpočetní API, ale také celkovou podobu programu, jeho vzhled, funkčnost a v neposlední řadě rozhraní mezi uživatelem a samotným programem. V následujících podkapitolách jsou tyto problémy probrány a zdůvodněny výběry daných řešení.

3.1 Specifikace požadavků

Implementace výše uvedených algoritmů v sekci 2.2.1 a 2.3.1 již existuje, a to od projektů implementuující základy, jako je například SPH 3D Real-Time Fluid Simulation [1], přes složitější projekty implementující několik různých přístupů k simulaci, například C++ Fluid Particles [2], až po komplexní řešení ve formě knihoven jako je SPHysics [3] či SPlisHSPlasH [4]. Při běžném hledání na internetu lze dokonce najít implementace běžící ve webovém prohlížeči s použitím javascriptového WebGL API. Mezi ně patří například WebGL Fluid Simulation [5] či Fluid Particles [6]. Z důvodu existence celé řady implementací, a to jak Eulerovy mřížkové metody, tak algoritmu Smoothed Particle Hydrodynamics, bylo nutné vybrat přístup k simulaci, případně jev, který není tak hojně představován. Z tohoto důvodu byl vybrán článek autorů Hochstetter a Kolb popisující vypařování a kondenzaci kapalin. Z důvodu komplexnosti celého tématu byl však vybrán pouze jeden jev, a to vypařování kapalin.

Hlavním požadavkem výsledné aplikace je schopnost odsimulovat tekutinu nacházející se v nějakém předem definovaném prostoru, vizualizovat současný stav kapaliny ve formě částic či objemu a možnost celý průběh simulace řídit. Uživatel má tedy přístup k ovládacím prvkům schopným simulaci pozastavit, krokovat či resetovat do výchozího stavu. Pro testování v rámci implementace, ale následně i používání aplikace, se jeví jako vhodné využít také ukládání současného stavu simulace a jeho případné opětovné načtení. Dalším požadavkem je možnost aplikace reprodukovat výslednou simulaci, a to z důvodů především prezentačních, ale i z důvodu rychlosti výpočtu simulace. Simulace jako takové jsou často

[1] https://github.com/saeedmahani/SPH-3D-Fluid-Simulation

[2] https://github.com/TroyZhai/CPP-Fluid-Particles

[3] https://wiki.manchester.ac.uk/sphysics/index.php/Main_Page

[4] https://github.com/InteractiveComputerGraphics/SPlisHSPlasH

[5] https://github.com/PavelDoGreat/WebGL-Fluid-Simulation

[6] https://github.com/dli/fluid

velmi náročné na výpočet a pro stabilní simulaci je třeba zvolit nízký krok simulace, což v kombinaci s náročnými výpočty může vést k výpočtu řádově jednotek snímků za vteřinu, případně několik vteřin či minut na snímek v případě složitějších simulací. Z toho důvodu je vhodné, aby aplikace mohla celý průběh simulace nahrávat a následně celý proces zobrazit v reálném čase.

Jak bylo však již zmíněno v úvodu, tato práce je zaměřena především na prozkoumání možností v oblasti simulace kapalin, načerpání nových informací a rozšíření okruhu znalostí. Z toho důvodu hlavním cílem této práce není přinést revoluční model simulace kapalin, přidat nějaký dříve opomíjený jev související s chováním kapalin, či snad poukázat na chyby v jiných modelech. Tato práce si klade za cíl proniknout do poměrně zajímavého okruhu simulace tekutin, implementovat a předvést její základy, ale také se zlepšit jak v běžném programování, tak v práci s grafickými výpočetními jednotkami a jejich využitím při provádění paralelních výpočtů. Výsledná aplikace tak nemá ambice býti „state-of-the-art“ produktem v oblasti výpočtů dynamiky kapalin, nicméně se spíše jedná o prezentaci nastudovaných materiálů a jejich aplikace v praxi.

Obrázek 3.1: **Existující řešení.** Na obrázku vlevo je znázorněna částicová simulace z knihovny SPlisHSPlasH. Vpravo pak lze videt výsledek z Eulerovy mřížkové metody z webové aplikace WebGL Fluid Simulation.
Zdroj: https://github.com/InteractiveComputerGraphics/SPlisHSPlasH
Zdroj: https://github.com/PavelDoGreat/WebGL-Fluid-Simulation

3.2 Propojení simulací

Tato práce se tedy zaměřuje mimo jiné na článek [20] autorů Hochstetter a Kolb z University of Siegen v Německu, který se snaží implementovat. Ačkoliv se s kondenzací a vypařováním lze setkat běžně každý den, v rámci simulace kapalin a počítačové grafiky není mnoho prací věnujících se tomuto tématu. Z tohoto důvodu se pomocí zmíněného článku snaží autoři představit novou metodu, jak takové změny skupenství simulovat. Autoři se rozhodli využít dvou metod simulace. Pro kapalné skupenství využili metodu Smoothed Particle Hydrodynamics (viz. kapitola 2.3.1), plynné skupenství pak bylo simulováno za využití mřížkové metody (viz. kapitola 2.2.1). Pro kondenzaci na povrchu pevných těles následně využili textur nesoucích potřebné údaje. Přestože se tato práce zabývá implementací pouze vypařování kapalin, je zde podrobněji celá metoda popsána a to včetně kondenzace. Celý popis pak vychází z článku napsaného autory. [20]

Základní princip

Popisovaná simulace se skládá ze tří částí, částicový systém pro simulaci kapalného stavu, mřížkový systém pro simulaci plynného stavu a textury udržující informace o množství kondenzace na pevných tělesech. Principy SPH simulace a mřížkové metody již byly popsány výše v kapitolách 2.3.1 a 2.2.1, avšak pro správné fungování cyklu vypařování a kondenzace je nutné do obou systémů zavést novou veličinu, a to teplotu.

U mřížkové metody se nejedná o žádný velký zásah. Jedná se o pouhé advekování teploty a hustoty pomocí pole vektorů rychlostí. Postup veličin polem je popsán následujícími rovnicemi 3.1 a 3.2.

$$\frac{\partial T}{\partial t} = -(\mathbf{u} \cdot \nabla)T \tag{3.1}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho \tag{3.2}$$

Pro zachování vlastnosti plynů, kdy horký plyn stoupá a těžký studený plyn klesá, je nutné zavést do systému vztlakovou sílu. Ta je popsána následující rovnicí 3.3 a je součástí parametru f rovnice 2.1,

$$f_{buoy} = -\alpha\rho\hat{\mathbf{g}} + \beta(T - T_{amb})\hat{\mathbf{g}} \tag{3.3}$$

kde α a β jsou předem definované konstanty a $\hat{\mathbf{g}}$ je normalizovaný vektor gravitačního zrychlení.

Pro SPH simulaci je nutné počítat pro každou částici také teplotu. Pro modelování teploty v rámci kapaliny využili autoři článku rovnici 3.4 vytvořenou dvojící Cleary a Monaghan [12],

$$\frac{\partial T}{\partial t} = \frac{V_1}{m_i C_i} \sum_j \frac{4k_i k_j}{k_i + k_j} V_j \frac{(T_i - T_j)}{||\mathbf{r}_i - \mathbf{r}_j||} \nabla W(\mathbf{r} - \mathbf{r_j}, h) \tag{3.4}$$

přičemž C_i označuje tepelnou kapacitu částice i a k_i tepelnou vodivost.

Nakonec jsou vytvořeny textury pro pevná tělesa. Každému tělesu náleží několik textur. Základní textura v každém texelu obsahuje množství zkondenzované kapaliny. Použitím textury a texelů menších než je částice z SPH simulace je docíleno jemnějšího detailu při kondenzaci. Dále je k dispozici přepočítaná textura obsahující hodnotu maximálního množství kapaliny v daném texelu a textura obsahující mapu pozic pro vytváření SPH částic. Další textury obsahují například teplotu, případně historii kondenzace.

Míra kondenzace a vypařování

Při vypařování a kondenzaci je nutné zjistit množství hmoty, které se transformuje z jednoho skupenství do druhého. Pro výpočet změny hmotnosti v povrchu kapaliny je nabízen následující vzorec 3.5 založený na empirických datech, [35]

$$\frac{\partial m_s}{\partial t} = A_s(a + b \cdot ||\mathbf{u}||)(p_s^{sat} - p_c) \tag{3.5}$$

kde A_s je plocha kontaktu kapaliny, a a b jsou konstanty ovlivňující vypařování, p_s^{sat} je tlak nasycených výparů u povrchu kapaliny a p_c je tlak výparu v buňce. Hodnoty tlaků jsou vypočítány následujícími rovnicemi 3.6 a 3.7 [41],

$$p_c = \frac{m_c R_w (T_c - 273,15°C)}{V_c} \tag{3.6}$$

$$p_s^{sat} = 611,2\text{Pa} \cdot \exp\left(\frac{17,62 \cdot T_s}{243,12°C + T_s}\right) \tag{3.7}$$

kde T_c značí teplotu buňky, T_s teplotu konkrétního surfelu a R_w je molární plynová konstanta.

Z rovnice 3.5 je získána pouze míra vypařování z povrchu kapaliny, nicméně míru kondenzace lze vyjádřit s pomocí stejné rovnice. V závislosti na znaménku výsledku dané rovnice lze následně určit, zda se jedná o kondenzaci či vypařování. Z toho vyplývají následující rovnice 3.8 a 3.9,

$$\textit{míra vypařování}(c,s) = \max\left(\frac{\partial m_s}{\partial t}, 0\right) \tag{3.8}$$

$$\textit{míra kondenzace}(c,s) = \min\left(\frac{\partial m_s}{\partial t}, 0\right), \text{kde } b = 0 \tag{3.9}$$

přičemž c značí buňku a s značí surfel, tedy povrchovou jednotku kapaliny. Pro zjištění hodnot buňky na konkrétní pozici $\mathbf{r}$ lze využít například trilineární interpolace hodnot buněk v okolí. Rovnice 3.5 přináší určitou závislost vypařování na proudění vzduchu v okolí, nicméně kondenzace v žádném případě nezávisí na okolní rychlosti vzduchu, proto jej lze z rovnice vyškrtnout za pomocí argumentu $b = 0$. Nakonec je nutné při každém přenosu zajistit, aby se z kapaliny odebralo stejné množství hmotnosti, jako je přijato do plynné fáze. Je tedy usilováno o zajištění zákonu o zachování hmotnosti a nezpůsobovat úbytek či přírůstek hmotnosti v celém systému simulace.

Simulace

Pro správné fungování simulace, přenosů tepla a hmotností je nutné veškeré systémy propojit. Je zapotřebí přenos teplot mezi kapalinou a výpary a texturami a částicemi. Pevná tělesa jsou reprezentována pomocí částic, a to z důvodu snadnější interakce kapalina $\leftrightarrow$ těleso nejen v rámci sil, ale i přenosu tepla. Zároveň musí docházet k přenosu hmotnosti mezi všemi třemi systémy. Celé propojení je znázorněné na obrázku 3.2.

Přenos tepla

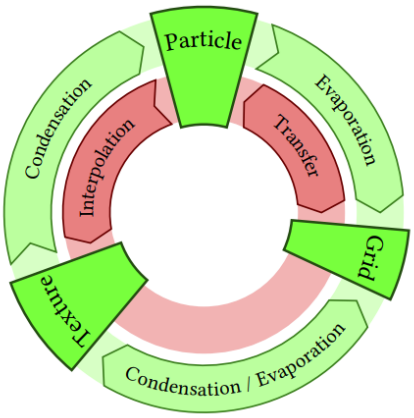
Pro fungování simulace je nutné vytvořit přenos tepla částice $\leftrightarrow$ buňka a jelikož se pevná tělesa sestávají z částic tak také částice $\leftrightarrow$ texel.

Je-li uvažován celý systém kapaliny a jejích výparů, je zřejmé, že z jedné buňky obsahující výpary může dojít k přenosu tepla do několika částic zároveň, a naopak z jedné částice může dojít k přenosu tepla do více buněk. Autoři článku tedy představují dvě rovnice, rovnici 3.10 pro přenos částice $\rightarrow$ buňka a rovnici 3.11 pro přenos buňka $\rightarrow$ částice.

$$\frac{\partial T_i}{\partial t} = \frac{1}{\rho_i C_i} A_i \sum_c \frac{4 k_i k_c}{k_i + k_c} (T_i - T_c) a_{ic} \tag{3.10}$$

$$\frac{\partial T_c}{\partial t} = \frac{1}{\rho_c C_c} A_i \sum_j a_{jc} \frac{4 k_j k_c}{k_j + k_c} (T_c - T_j) A_j \tag{3.11}$$

Přenos tepla do textury je následně proveden pouhou interpolací hodnot z částic pevného tělesa do jednotlivých texelů. Tímto jednoduchým způsobem dostane každý texel vlastní teplotu, čímž není nutné aplikovat složité vzorce na výpočet a zvyšovat tím tak výpočetní náročnost.



Obrázek 3.2: **Propojení tří systémů simulace.** Obrázek znázorňuje propojení tří systémů simulace. Těmito systémy jsou částice (Particle), textura (Texture) a mřížka (Grid). Zelený kruh zobrazuje přenos hmotnosti a červený kruh přenos teploty. Přesněji červený kruh ukazuje, zda se jedná o přenos (Transfer) či interpolaci (Interpolation). V kruhu lze následně vidět, mezi jakými částmi systému dochází k jakému typu přenosu hmotnosti, zda se jedná o vypařování (Evaporation) nebo kondenzaci (Condensation). **Zdroj:** Evaporation and condensation of SPH-based fluids [20]

Přenos hmotnosti

Přenos mezi texturou a částicemi probíhá pouze jedním směrem, a to texel → částice pomocí kondenzace. Pokud se nachází na textuře v okolí pozic pro generování částic dostatečná hmotnost kondenzované kapaliny, dochází ke generování částice. Dané množství kapaliny je odebráno z textury a je vytvořena částice o stejném množství kapaliny.

Výměna hmotnosti mezi texely a buňkami s výpary je poměrně jednoduchá. Základem je výpočet míry vypařování a kondenzace pomocí vzorců 3.8 a 3.9. Pokud bychom však slepě aplikovali tyto vzorce, mohlo by dojít k odebrání většího množství hmotnosti kapaliny než je přítomné v buňce či texelu, případně přesaturování, kdy by bylo obsaženo více hmotnosti, než je povolené množství. Z tohoto důvodu musí být všechny přenosy škálovány podle množství již přítomné kapaliny či výparů.

Pro vypařování a kondenzaci mezi částicemi kapaliny a buňkami platí stejný postup jako mezi texely a buňkami. Musíme dodržet stejná pravidla, nicméně u částic nedochází ke změně hmotnosti, nýbrž se mění jejich škálovací faktor. Tento faktor pak ovlivňuje veškeré vlastnosti částice, jako je tlak, hustota apod. až na hmotnost, která jím není ovlivněna. V celé simulaci se snažíme dodržet, aby každá částice měla stejný faktor $w_i = 1$, proto pokud nalezneme částice s nízkými faktory, můžeme dané částice sloučit a vytvořit jednu s vyšším faktorem.

3.3 Výběr grafického/výpočetního API

Výběr vhodného API pro práci s grafickou kartou je velmi důležitým krokem a při výběru záleží na několika faktorech, a to především na snadnosti použití, zda je daný framework multiplatformní, či jaké přináší možnosti pro práci s grafickou kartou obecně. Níže jsou pak vyjmenováni hlavní zástupci a jsou popsány jejich hlavní výhody a nevýhody.

DirectX

DirectX je kolekce různých API využívaných především pro tvorbu her a práci s multimédii. Nejznámější komponentou je nejspíše Direct3D, starající se o vykreslování 3D grafiky, nicméně mezi dalšími komponentami lze najít například Direct2D, jak již název napovídá, pro vykreslování 2D grafiky, DirectCompute pro výpočetní účely, XAudio2 pro nízkoúrovňovou práci se zvuky a v neposlední řadě také DirecX Raytracing pro raytracing v reálném čase, případně DirectML pro strojové učení na grafických kartách. [2]

Kompletní kolekce DirectX je vyvíjena firmou Microsoft a je využívána především v počítačích se systémy Windows a v herních konzolích Xbox. V době tvorby této práce není tento balíček nativně podporován v Linuxových systémech a pro spouštění DirectX aplikací je nutné využívat mezivrstev, jako je například Wine. Ve verzi 12 je pak Direct3D orientován na mnohem nižší úrovni podobně jako Vulkan a je na programátorovi, aby se postaral o mnohem více věcí, než tomu bylo v předchozích verzích. Tím se stává mnohem komplexnějším nástrojem umožňujícím jemnější práci s grafickou kartou. Pro tvorbu shaderů se pak využívá jazyka HLSL (High-Level Shading Language) vyvinutého speciálně pro Direct3D verze 9 a vyšší. [33] [16]

OpenGL

OpenGL je alternativa k Direct3D, oproti DirectX se jedná o multiplatformní API schopné spouštění na nespočtu systémů, čímž při správném postupu lze docílit vysoké přenositelnosti programu. API je vyvíjeno a spravováno konsorciem Khronos Group, a jak z názvu vyplývá, je OpenGL standard distribuovaný v Open-source licenci, tudíž zkušení programátoři mohou nahlédnout do zveřejněných specifikací a lépe pochopit jeho funkčnost. Pro OpenGL mluví i jeho mutace ve formě OpenGL ES a WebGL. OpenGL ES je API pro práci s 2D a 3D grafikou na embedded systémech a mobilních zařízeních. Celkově se jedná o odlehčené verze OpenGL poskytující podmnožinu funkcí dané specifikace. V praxi to znamená, že například kódy a shadery napsané v OpenGL ES 3.0 lze bez problému spustit i pod OpenGL 4.3 a naopak, pokud se používají pouze funkce nacházející se i v dané verzi pro embedded systémy. WebGL je pak JavaScriptové API pro vykreslování 2D i 3D grafiky ve webových prohlížečích, je postavené na OpenGL ES a pro shadery využívá stejné specifikace, a to GLSL ES (OpenGL ES Shading Language). [3] [5] [17]

OpenGL jako takové je jazyk vyšší úrovně, což z něj dělá ideální jazyk do začátků díky jeho relativní jednoduchosti. To umožňuje programátorovi pomocí několika málo řádků vykreslit jednoduchý trojúhelník. Nespornou a nejspíše i největší výhodou tohoto API je tedy jeho relativní jednoduchost, ale zároveň také dokumentace. Jedná se o velmi rozšířené API, proto lze nalézt celou řadu různých učebních materiálů. Přestože je nutno si dát pozor na jejich aktuálnost, lze stále naleznout celou řadu užitečných návodů a rad. S nevýhodou jednoduchosti přichází však omezení vysoko-úrovňového jazyka, a to možnost do detailu ovládat grafickou jednotku a spravovat její paměť. Zmíněné omezení nemusí vyloženě znamenat problém, existuje několik úspěšných her vytvořených na tomto API, nicméně pro

účely maximalizování grafického či výpočetního výkonu je vhodnější využít spíše nízkoúrovňové API, jako je například Vulkan či již zmíněný Direct3D 12.

OpenCL a CUDA

OpenCL je framework pro vytváření programů schopných paralelizace, podobně jako je tomu u výpočetních shaderů. Princip fungování je velmi podobný, programátor vytvoří nejprve zaváděcí program běžící na CPU, kde vytváří a spravuje potřebné prostředky. Dále je také nutné, aby napsal tzv. kernel, který běží na daném výpočetním jádře a je schopný paralelizmu stejně jako u výpočetních shaderů. Výhodou OpenCL je, že tyto kernely mohou běžet na téměř jakémkoliv hardware. OpenCL programy lze spouštět napříč heterogeními platformami a lze je spouštět nejen na klasických grafických kartách, ale také na běžných procesorech, či specializovaných zařízeních jako jsou FPGA nebo DSP čipy. Další nespornou výhodou je využití mnohem volnějšího jazyka pro programování kernelů, existuje totiž komunitní mutace jazyka C++17 podporovaná i v OpenCL kernelech. Nejedná se o kompletní podporu a některé možnosti jazyka nejsou podporovány, ale oproti GLSL má programátor k dispozici mnohem flexibilnější nástroj. [40]

CUDA je pak podobný čistě výpočetní framework od firmy Nvidia. Jazyk je postaven opět na jazyku C++ a celkově se jedná o velice snadný nástroj co se týče jeho samotného používání. Funkce běžící na CPU a GPU mohou být psány do jednoho souboru a jsou pouze opatřeny speciálním klíčkovým slovem. Největším problémem tohoto frameworku však je, že je funkční pouze na CUDA jádrech grafických karet firmy Nvidia a je nemožné tyto programy spouštět na jiném hardware. [9] [10]

Celkově jsou tyto nástroje velice vhodné pro výpočetní problémy, především problémy, které lze paralelizovat pro snížení výpočetní doby. OpenCL nabízí nespornou výhodu v možnosti využití různých součástí systému a lze tak snadněji rozložit práci. Méně náročnější práci pro malý počet vláken může vykonávat procesor, zatímco náročnější výpočty může vykonávat grafická karta, případně specializovaný hardware uzpůsobený na konkrétní problémy. CUDA trpí na úzkou podporu zařízení, ale naopak vyhrává na poli jednoduchosti použití. Oba dva frameworky jsou však zaměřené čistě na výpočty a jejich jazykové dialekty neobsahují grafické funkce jako je tomu u jazyků shaderů OpenGL, Direct3D a podobných. Chybí také grafický výstup a je nutné si výsledky výpočtů pro zobrazení předávat skrze procesor, což výrazně zpomaluje celý chod aplikace. Existuje i možnost interoperability mezi například OpenCL a OpenGL, tedy ponechání dat na grafické kartě a využití jiným API. Celkově se však může jednat o vcelku složitý proces, především v rámci synchronizace a zachování integrity dat.

Vulkan

Vulkan je nejmladší ze zmíněných API, jedná se o produkt již zmíněných Khronos Group. Celé API bylo vystavěno na základech API Mantle od AMD, které bylo darováno, aby měli vývojáři z Khronos Group základy, na kterých stavět. Vulkan je nízko-úrovňové API s malou režií podobně jako Direct3D 12, ale zároveň je mulltiplatformní podobně jako OpenGL, tudíž produkty využívající tohoto API lze spouštět na různých systémech. Vulkan stejně jako OpenGl podporuje výpočetní shadery. Díky své explicitnosti je programátor využívající Vulkan nucen nastavit mnohem více věcí než by musel v OpenGL a ačkoliv má tak nad celým psaným programem mnohem větší kontrolu, program jako takový se stává mnohem komplexnější. Jednoduché vykreslení trojúhelníku tak zabere mnohem více práce a řádků kódu, než by tomu bylo například ve zmiňovaném OpenGL. Mezi další výhody

patří například jednotné API, či mnohem větší kontrola nad alokovanou pamětí. Zatímco je OpenGL rozděleno na verzi pro embedded systémy a běžnou plnou verzi, Vulkan nedělá rozdíl, a pokud vyjde určitá funkčnost na stolní zařízení, je automaticky dostupná i na mobilních zařízeních. [34] [18]

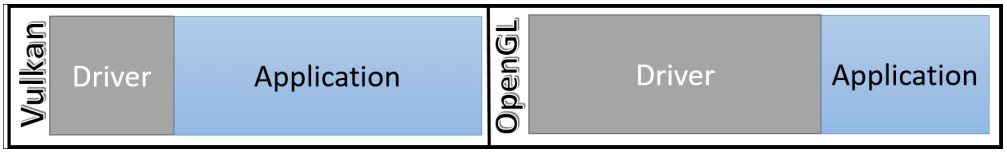
3.3.1 Výběr

Jako první byl zavrhnut framework OpenCL. Ačkoliv se jedná o velmi robustní systém schopný poskytnout výpočetní výkon grafické karty, neschopnost poskytnout grafický výstup, bez složitějších postupů, byl rozhodující faktor pro vyřazení. Nepomohly ani nízké zkušenosti s daným frameworkem a malá pravděpodobnost využití možností typických pro OpenCL, jako je například použitelnost nad různými typy hardwaru. Z podobných důvodů byl vyřazen i CUDA framework. Ačkoliv při vývoji byla využita grafická karta Nvidia a karty této firmy jsou poměrně rozšířené, dochází k omezení platforem, na kterých lze výsledný program spustit.

Po tomto zúžení kandidátů bylo jasné, že se bude tedy jednat o grafické API schopné využívat grafických čipů k náročným paralelním výpočtům. Dalším vyřazeným kandidátem byl DirectX, respektive Direct3D. Důvod je velice podobný jako u CUDA Frameworku. Jelikož byl celý projekt implementován z valné většiny v systému Linux, bylo by krajně nepraktické využívat pro implementaci mezivrstev zajišťujících kompatibilitu, zvláště když kompatibilita není stoprocentní a s velkou pravděpodobností by bylo nutné kód ohýbat, aby byl vůbec spustitelný.

Poslední rozhodování bylo mezi API OpenGL a Vulkan. OpenGL je jednodušší, více zdokumentované a především nevyžaduje tolik přípravy před samotným vykreslováním, případně výpočtem. Vulkan má však nespornou výhodu při práci s pamětí a hlášením chyb. Při alokaci paměti musí programátor explicitně specifikovat její účel, zda-li se jedná o datový buffer, či například buffer s vrcholy modelu, ale také příznaky přístupu do paměti, tedy jestli se bude z dané paměti kopírovat, zda bude paměť viditelná jen pro dané zařízení, nebo bude možné k ní přistupovat i ze strany procesoru. Takové nastavení pak může mít vliv na výkon a při správném výběru příznaků lze zrychlit přístup do paměti. U Vulkanu je také zajímavé řešení chybových stavů či nastavení. OpenGL neustále kontroluje, zda nenastala chyba, tedy i v případě, že máme aplikaci, ve které nenastávají chyby. Neustálá kontrola chyb tak může ovlivnit výkonnost výsledné aplikace. Naopak Vulkan ve výchozím nastavení neprovádí žádné kontroly a pro hlášení chyb je nutné zapnout tzv. validační vrstvy. Validační vrstvy jsou části kódu mezi aplikací a samotným Vulkan API, schopné kontrolovat správnost volaných funkcí. Zkušení programátoři si pak mohou vytvořit vlastní vrstvy, kontrolující specifické stavy. Na obrázku 3.3 lze pak porovnat nutnost velikosti aplikace u Vulkanu a OpenGL. Zatímco u OpenGL se o hodně věcí stará samotné API/ovladač a na programátora je tedy kladen mnohem menší nárok, u Vulkanu je vyžadována mnohem větší správa přímo v samotné aplikaci. [34] [32]

Z těchto důvodů byl tedy vybrán Vulkan. Jedná se o nástroj umožňující větší kontrolu nad grafickým čipem, ale zároveň se jedná o jakéhosi nástupce OpenGL v grafických API. Dalším rozhodujícím faktorem při výběru byly zkušenosti s daným API. V průběhu studia na Fakultě informačních technologií bylo zapotřebí vypracovat několik projektů s využitím OpenGL, čímž byly získány určité zkušenosti. Na druhé straně je celá tato závěrečná práce zaměřená především na studium nových poznatků a získávání zkušeností. Tento fakt tedy přispěl k výběru zmíněného Vulkanu, jelikož bylo uznáno za vhodné při zkoumání nových postupů zakomponovat i zkoumání nových technologií.



Obrázek 3.3: **Porovnání aplikací Vulkan a OpenGL.** Obrázek představuje dvě aplikace se stejnou funkčností, jednu napsanou pro Vulkan a druhou pro OpenGL. Šedá část aplikace značí kód ovladače, tedy úsek, o který se programátor nemusí starat a je řešen bez nutnosti zásahu. Modrá část následně představuje kód samotné aplikace, kterou musí programátor vytvořit. Při porovnání nízkoúrovňového API jako je Vulkan a API vyšší úrovně, jako je OpenGL, je znatelné, že u nízkoúrovňových API je na programátora kladen mnohem větší nárok co se týče množství psaného kódu.
Zdroj: Learning Vulkan [34]

3.4 Uživatelské rozhraní

Uživatelské rozhraní je velice důležitou součástí programu. Existuje celá řada způsobů, jak přistoupit k uživatelskému rozhraní. Je možné využít pouze běžné komponenty jako jsou tlačítka, rolovací menu či textová pole. Další možností je využití prostého příkazového řádku, konfiguračních souborů, či vymyslet úplně vlastní jedinečný způsob komunikace uživatele s programem. Dalším důležitým prvkem, zvláště u grafických aplikací, je vstupní zařízení. Lze zvolit ovládání aplikace pouze za pomoci klávesnice, využít i myši, případně je možné využít různých více či méně specializovaných ovladačů. Alternativou je i zakomponování netradičních a běžně nepoužívaných vstupních metod, jako jsou například hlasové příkazy, gesta či různé senzory nebo kamery zpracovávající dění v určitém prostoru.

Obrázek 3.4: **Dear ImGui.** Demostrační aplikace knihovny `Dear ImGui`. Na obrázku jsou vidět různé komponenty, které lze využít pro tvorbu uživatelského rozhraní. Knihovna poskytuje běžné komponenty jako například okna, tlačítka, textová pole, ale také méně tradiční, nicméně pro grafické aplikace užitečné komponenty jako výběry barev nebo grafy.
Zdroj: https://github.com/ocornut/imgui

Pro simulaci se jeví jako nejvhodnější použití klávesnice a myši a běžného rozhraní sestávajícího z oken, menu, tlačítek a podobných komponent. Nicméně současně s grafickým uživatelským rozhraním je vhodné mít i možnost konfigurace ze souboru. Soubor je tak možné využít pro načítání výchozích hodnot do příslušných polí v grafickém rozhraní, což usnadňuje počáteční konfiguraci a odstraňuje nutnost vždy vyplňovat veškeré konstanty

před začátkem každé simulace. Tento přístup zároveň poskytuje možnost využívání konfiguračních souborů pro tvorbu různých scénářů. V závislosti na obsažených datech je následně možné různě vyplněné konfigurační soubory využívat i k tvorbě rozličných simulačních scénářů. Uživatel takto dostává možnost relativně snadno přepínat mezi různými konfiguracemi a zkoumat chování v různých situacích.

Grafické rozhraní samozřejmě musí obsahovat prvky pro kontrolu simulace, změnu parametrů různých částí simulace a obecně je s ním uživatel schopný ovládat aplikaci a využívat veškerou funkčnost popsanou v podkapitole 3.1. Při práci s aplikací je pro uživatele užitečné zaměřit se na jednotlivé místa v simulaci, tudíž mít možnost pohybovat se v prostoru, přibližovat a oddalovat simulační prostor, tedy obecně nějakým způsobem ovládat kameru. Mezi další, běžně ne tolik důležité prvky uživatelského rozhraní simulace, lze zařadit například možnost kontroly pozice světla osvětlující celý prostor, případně změna barvy simulované kapaliny. Zmíněná změna barvy se může zdát jako bezvýznamná funkce pro celou aplikaci, nicméně zlepšuje celkový dojem z aplikace a zároveň může pomoci při identifikaci různých scénářů. Uživatel může vytvářet nahrávky simulací různých scénářů, kde každý scénář má jiné výchozí parametry, jako teplotu, viskozitu, případně velikost kroku simulace. Každý nahraný video soubor je následně generován s určitým výchozím jménem, nicméně moderní operační systémy poskytují často náhled videa jako ikonu souboru a zde má uživatel možnost odlišit jednotlivé nahrávky. Například modrá kapalina simulující méně viskózní kapalinu podobnou vodě, případně žlutá kapalina připomínající spíše olej či med.

Kapitola 4

Implementace

V následujících kapitolách je řešena stránka implementace celého simulátoru, který je jádrem této práce. Jsou zde popsány nejen použité knihovny a nástroje, ale také jednotlivé části celého programu. Nejprve je zde popsána zjednodušená posloupnost programu, kde je vysvětlen základní tok programu. Následně jsou probrány jednotlivé části programu, jejich funkčnost a návaznost na ostatní části. Většina algoritmů však není probírána dopodrobna a čtenář je odkázán na vysvětlené algoritmy v rámci kapitoly 2 Teorie. Popsána je pak pouze implementace algoritmů, která nemusí být z teoretické kapitoly jasná, případně teorii nějakým způsobem modifikuje či doplňuje.

4.1 Použité knihovny

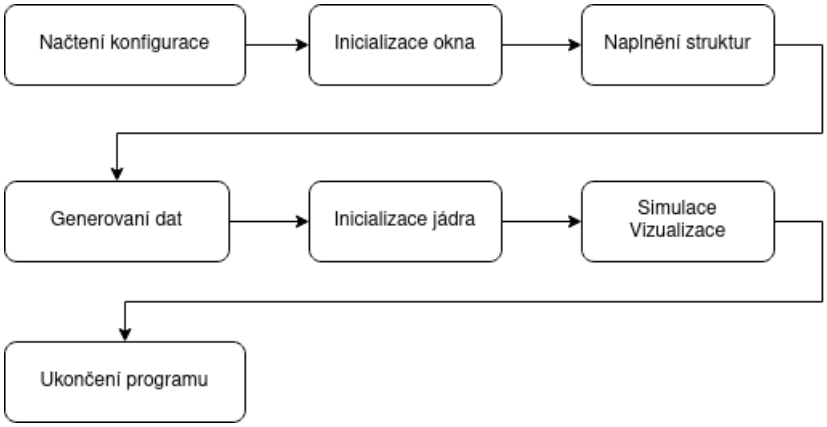
Pro implementaci byl zvolen jazyk `C++` ve standardu 20. Pro zjednodušení a automatizaci překladu pak bylo využito nástrojů `CMake` [1] a `Ninja` [2]. Samotné psaní kódu pak probíhalo ve vývojovém prostředí `CLion` [3], přičemž tvorba shaderů probíhala v programu `Visual Studio Code` [4] s pomocí pluginu `GLSL Lint` [5]. Pro správu většiny knihoven třetích stran bylo využito nástrojů `Hunter` [6] a `CPM.cmake` [7].

- glm [8]
- GLFW [9]
- fmt [10]
- spdlog [11]
- stb [12]
- libshaderc [13]
- toml11 [14]
- range-v3 [15]
- tinyobjloader [16]
- camera [17]
- ffmpeg [18]

[1] https://cmake.org/
[2] https://ninja-build.org/
[3] https://www.jetbrains.com/clion/
[4] https://code.visualstudio.com/
[5] https://marketplace.visualstudio.com/items?itemName=CADENAS.vscode-glsllint
[6] https://hunter.readthedocs.io/
[7] https://github.com/TheLartians/CPM.cmake
[8] https://glm.g-truc.net/
[9] https://glfw.org/
[10] https://github.com/fmtlib/fmt
[11] https://github.com/gabime/spdlog
[12] https://github.com/nothings/stb
[13] https://github.com/google/shaderc/tree/main/libshaderc
[14] https://github.com/ToruNiina/toml11
[15] https://github.com/ericniebler/range-v3
[16] https://github.com/tinyobjloader/tinyobjloader
[17] https://learnopengl.com/Getting-started/Camera
[18] https://ffmpeg.org/

4.2 Základní struktura

Na obrázku 4.1 je zobrazena hlavní struktura programu. Jedná se především o inicializaci celé aplikace, kde samotná simulace představuje jediný blok. Každý blok v diagramu tvoří důležitou součást celé aplikace a je mu tedy věnována podrobněji vlastní podkapitola.



Obrázek 4.1: **Základní tok programu.** Prvním krokem v programu je načtení konfiguračního souboru a inicializace okna. Dalším krokem je naplnění potřebných struktur a generování dat, jako jsou například výchozí pozice částic. Data jsou v simulaci rozdělena na struktury nesoucí konkrétní informace o stavu simulace a na statické balíčky dat obsahující potřebné konstanty a údaje o simulaci. Více jsou tyto struktury rozebrány v kapitole 4.5. Následuje inicializace jádra, které obsahuje uživatelské rozhraní a veškeré moduly pro simulaci a vykreslování. Po inicializaci jádra lze přistoupit k samotné simulaci a vizualizaci. V tomto bloku se program drží tak dlouho, dokud není uživatelem ukončen.

Aplikace začíná načtením konfiguračního souboru a inicializací logovacího systému. Konfigurační soubor obsahuje především většinu konstant a parametrů pro správný chod simulace, ale také některé informace týkající se aplikace jako takové. Dalším krokem je inicializace okna schopného zobrazovat simulovaná data převedena do vizuální podoby. Následuje naplnění simulačních struktur a generovaní simulačních dat. Simulační struktury jsou balíčky obsahující statické informace o simulaci, kdežto simulační data obsahují popis konkrétních částí simulace, přesněji částic či buněk. Tato data pak přebírá jádro, které se stará o samotnou simulaci a vizualizaci. Jádro v sobě skrývá nejen moduly pro jednotlivé části simulací, ale zároveň moduly pro vizualizaci výsledků a grafického uživatelského rozhraní. Po inicializaci přichází na řadu samotná simulace a vizualizace. Oba tyto moduly běží v nekonečné smyčce, přičemž vizualizace probíhá každý cyklus, ale výpočty simulace pouze v případě, že je uživatelem simulace spuštěna. Celá tato smyčka trvá tak dlouho, dokud nedojde k ukončení programu uživatelem. V následujících kapitolách jsou jednotlivé části podrobněji rozebrány.

4.3 Konfigurace

Jak je ukázáno na obrázku 4.1, prvním hlavním krokem celého programu je načtení konfiguračního souboru. Samotná struktura konfiguračního souboru je pak popsána v příloze B, nicméně stručně řečeno soubor je rozdělen na několik částí, kde každá část má svůj určitý význam v celém celku programu. Velkou většinu souboru tvoří především parametry

simulace, rozdělené do tří bloků, tedy mřížková metoda, metoda Smoothed Particle Hydrodynamics a nakonec samotné vypařování. Zbývající bloky se pak týkají nastavení samotné aplikace, jako jsou rozměry okna, cesta k shaderům či například poslední pozice a úhel kamery.

Existuje celá řada formátů využívaných pro konfigurační soubory, běžný `INI`, komplexnější `YAML`, případně i formáty primárně neurčeny pro konfiguraci, jako je třeba `JSON` nebo `XML`. Pro účely tohoto projektu byl zvolen formát `TOML`. Jedná se o relativně nový formát, který se snaží být pro člověka čitelný, minimalistický a zároveň je vystaven tak, aby byl schopný se snadno mapovat do hašovací tabulky [39]. Tento formát byl vybrán především kvůli své zvýšené čitelnosti oproti například formátům `JSON` či `XML`, a to především protože byly očekávány častější změny v souboru a chtěla být poskytnuta možnost vyčíst ze souboru současné nastavení simulace. Těchto možností bylo využito především v prvotních fázích vývoje, kdy nebyla všechna nastavení propojena do uživatelského rozhraní.

Samotné načtení konfiguračního souboru je pak vcelku primitivní záležitostí. K snadnější manipulaci se souborem byla využita knihovna `toml11`, která poskytuje funkce jak pro načítání, tak ukládání dat ve formátu `TOML`. Načtená data jsou následně uložena do třídy `Config` obsahující struktury `AppConfig` a `VulkanConfig`, kde část `AppConfig` je dále rozdělena na struktury dle jednotlivých simulací. Samotná instance třídy `Config` je různě předávána mezi ostatními částmi programu dle potřeby, přičemž její obsah je využíván k nastavování programu a tvorbě výchozího stavu simulace. Do samotného souboru nejsou propisovány žádné změny udělané v parametrech simulace během používání aplikace. Výjimkou je pouze pozice kamery, která je při ukončení uložena do příslušného souboru, takže po opětovném spuštění aplikace se stejným konfiguračním souborem dojde mimo jiné k obnovení poslední pozice kamery, a uživatel nemusí opět měnit kameru do vhodnější pozice pro daný scénář.

4.4 Okno

Jelikož výsledkem této práce je grafická aplikace, schopná vizualizovat počítané výsledky uživateli, je nutné vytvořit okno, které umožňuje zobrazovat vykreslovaný výstup z použitého API Vulkan. Pro tento účel bylo využito knihovny `GLFW`, schopné vytvářet okna a plochu k vykreslování nejen pro Vulkan využívaný v této práci, ale také OpenGL a OpenGL ES. Dalšími užitečnými vlastnostmi, které `GLFW` poskytuje, je možnost využití různých vstupů. `GLFW` umožňuje využívat například v simulátoru požadovanou klávesnici a myš, ale zároveň různé časované události či události samotného okna. Tyto události pak může daná aplikace zpracovávat ve formě tzv. callbacků nebo pollingu.

V aplikaci byla vytvořena vlastní třída `GlfwWindow` zapouzdřující funkčnost samotného okna. Kromě poskytování samotného handelu okna pro případné ostatní třídy, obsahuje také funkce pro poskytování informací o okně jako jsou jeho rozměry. Hlavní částí je však implementace návrhového vzoru observer. Toto je zabezpečeno děděním od třídy `EventDispatchingWindow`, která poskytuje funkce pro registraci několika druhů funkcí zpracovávajících události daného okna. Přesněji se jedná o události pocházející z klávesnice a myši. Při registraci je funkce uložena do fronty a zároveň je vrácena instance třídy schopná daného posluchače událostí z fronty odstranit. Když následně dojde k vyvolání určitého typu událostí, jsou všechny funkce přítomné ve frontě zavolány a jako parametr je těmto funkcím předána zpráva obsahující důležitá data o události. Příslušná zpráva následně obsahuje data, jako je například typ klávesy či typ provedené akce. Tyto zprávy

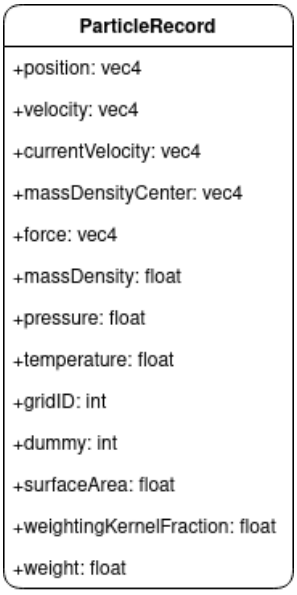
má za úkol daná funkce zpracovat a adekvátně na ně reagovat například změnou pozice či natočením kamery.

4.5 Simulační struktury a simulační data

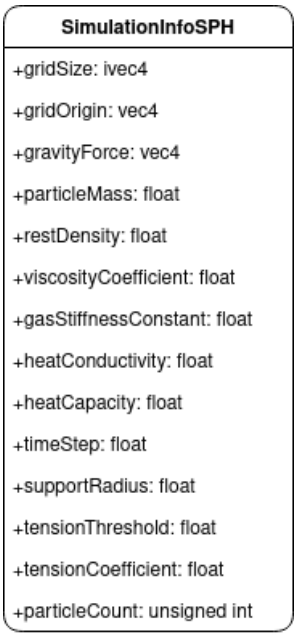
Pro správné fungování simulace je potřeba nejprve vyplnit několik struktur a inicializovat výchozí hodnoty pro částice a mřížkovou metodu. Struktury mající v názvu „Info“ jsou pak statické struktury, které se v čase samy nemění, a mění se pouze na popud uživatele, který zasahuje do parametrů simulace. Zbytek jsou datové struktury držící stav simulace a měnící hodnoty v jejím průběhu.

Smoothed Particle Hydrodynamics

Stav jednotlivých částic představuje pole sestávající se z několika struktur popsaných na obrázku 4.2a. Výchozí pozice pro částice jsou generovány v závislosti na objemu kapaliny, velikosti daného modelu a počáteční pozici. Důležité je upozornit, že generování nebere v potaz pozici a velikost simulovaného prostoru, a proto při špatném konfiguračním souboru může dojít k destabilizaci simulace. Níže budou popsány pouze komponenty, u nichž nemusí být na první pohled zřejmé, co představují.



(a) **Struktura představující částici**



(b) **Informační struktura**

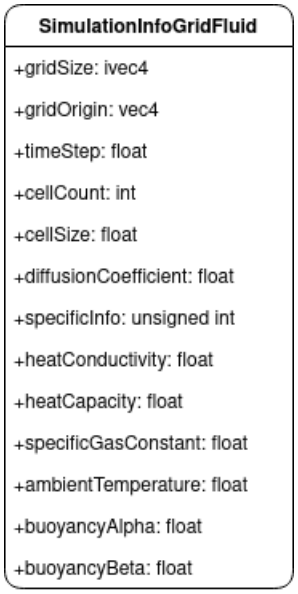
Obrázek 4.2: **Datové struktury pro SPH simulaci.** Na obrázku 4.2a je zobrazena struktura představující jednu částici. Většina položek je celkem intuitivně rozklíčovatelná, nicméně stojí za to zmínit přítomnost duplicitních rychlostí, kde položka `velocity` představuje rychlost v čase $t - \frac{1}{2}$ a `currentVelocity` představuje rychlost v čase t. Za zmínku stojí i položky `gridID` označující příslušnost k buňce a `dummy` představující položku využívanou především pro diagnostický výstup. Obrázek 4.2b představuje statická data neměnná v čase. Tato data slouží pro předávání informací, které se během simulace nemění, tudíž je není nutné ukládat u každé částice.

Struktura obsahuje dvakrát hodnotu rychlosti, nicméně nejedná se o duplicitní hodnoty, ale přítomnost obou dvou vyplývá z implementace algoritmu Leap-frog popsaném v kapitole 2.3.1. V tomto algoritmu dochází k střídavému výpočtu mezi rychlostí a pozicí. Při výpočtu pozice se vždy vychází z pozice v čase $t-1$ a výsledkem je pozice v čase t. U rychlosti ale nastává problém. Ta vychází z rychlosti v čase $t-\frac{1}{2}$ a výsledkem je rychlost v čase $t+\frac{1}{2}$. Z toho vyplývá, že z důvodu dodržení konzistentnosti je zapotřebí pro další výpočty využívat rychlost v čase t, tedy parametr `currentVelocity`, nicméně pro výpočet následující rychlosti $\mathbf{u}_{t+\frac{1}{2}}$ je nutné vycházet z předešlé rychlosti $\mathbf{u}_{t-\frac{1}{2}}$ reprezentované částí `velocity`. `MassDensityCenter` a `weightingKernelFraction` jsou pomocné parametry sloužící k předávání informací mezi jednotlivými částmi simulace. Těchto parametrů je využíváno pro výpočet povrchu, který částice zaujímá v celkovém objemu. Část `dummy` pak představuje hodnotu především pro testovací účely, schopnou ukládat různá data, a položka `gridID` značí příslušnost částice k buňce v rámci urychlení vyhledávání sousedů.

Informační struktura 4.2b je pak vcelku zřejmá a většina parametrů je naplněna z konfiguračního souboru. Za zmínku snad stojí pouze výpočet vyhlazovacího poloměru, který vychází z následujícího vzorce 4.1,

$$h = \sqrt[3]{\frac{3V\ x}{4\pi n}} \tag{4.1}$$

kde x představuje přibližný počet částic obsažených v poloměru a n je celkový počet částic.



Obrázek 4.3: **Informační struktura pro mřížkovou metodu.** Tento obrázek představuje statickou informační strukturu pro mřížkovou metodu. Zde stojí za zmínku především položka `specificInfo`. Tato položka nese specifické informace v závislosti na prováděném kroku mřížkové simulace. Hlavní využití má při paralelní verzi numerického algoritmu Gauss-Seidel. V tomto případě nese informace, o jaký prováděný krok se jedná a která iterace se právě počítá.

Mřížková metoda

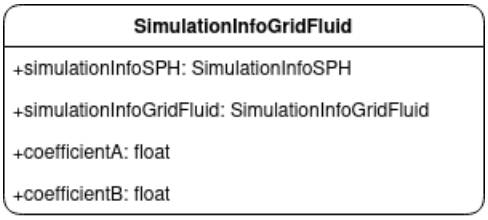
Mřížková metoda má ve strukturách minimum hůře pochopitelných částí. Jednou z nich je však parametr `specificInfo`, který je vytvořen tak, že v závislosti na současném průběhu

simulace nese jiné informace. Reálně je však využit pouze při výpočtu difuze a divergence. V těchto krocích je totiž předpokládáno využití Gauss-Seidlova numerického algoritmu. Ten je však nevhodný pro paralelní systémy, a proto je využito paralelní verze toho algoritmu. Zmíněný parametr `specificInfo` následně nese specifické informace potřebné pro daný algoritmus a zároveň udává informaci, zda se jedná o výpočet divergence nebo difuze. Parametry `bouyancyAlpha` a `bouyancyBeta` pak představují konstanty α a β ve vzorci 3.3.

Samotná data nesoucí stav simulace jsou uložena v samostatných polích. Existuje tedy pole nesoucí rychlosti v daných buňkách a pole nesoucí informace o hustotě a teplotě dané buňky. K tomuto rozdělení došlo především z důvodu mnoha kroků simulace a k častému užití pouze části dat. Obě pole jsou zároveň duplikovány, a to z důvodu nutnosti využití předešlých hodnot při výpočtu nových hodnot.

Vypařování

Jelikož vypařování funguje na principu propojení obou typů simulací, je logické, že v informační struktuře zobrazené na obrázku 4.4 jsou obsaženy informace jak o částicové simulaci, tak o mřížkové. Parametry `coefficientA` a `coefficientB` představují konstanty ve vzorci 3.5 ovlivňující rychlost vypařování. Jelikož simulace vypařování pracuje jak s částicemi, tak s buňkami, je nutné samozřejmě využít i pole částic z částicové simulace a pole rychlostí, hustot a teplot z mřížkové simulace.



Obrázek 4.4: **Informační struktura pro simulaci vypařování.** Informační struktura pro simulaci vypařování je kombinací dvou předešlých struktur, což vyplývá z faktu, že se jedná o propojení obou předešlých simulací. Jedinými novými parametry jsou pak koeficienty ovlivňující míru vypařování.

4.6 Jádro

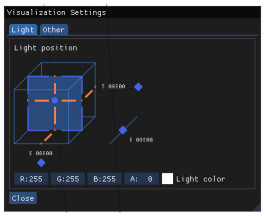
Jádro simulace je hlavním středobodem celé aplikace. Jedná se o třídu `VulkanCore`, která se stará nejen o inicializaci jednotlivých simulačních modulů, ale také o jejich správné spouštění, synchronizaci a inicializaci celého Vulkan API. Tato třída byla prvotně implementována na základě návodu [19]. Za pomoci zmíněného návodu byla vytvořena prvotní kostra aplikace a došlo k seznámení s API Vulkanu. K samotné implementaci však bylo využito rozhraní v jazyce C++, místo v návodu zmiňovaném C. Hlavičkový soubor `vulkan.hpp` totiž oproti jeho C verzi přináší několik nesporných výhod, kde mezi největší lze zařadit využití výčtových typů místo maker a využití `UniqueHandle` pro automatickou správu objektů.

V průběhu tvorby tohoto jádra následně docházelo k postupnému přesouvání důležitých komponent do vlastních tříd s tím, že každá třída dostala několik funkcí pro snadnější manipulaci s jejími součástmi. Navíc pro složitější komponenty, jako například `Pipieline`,

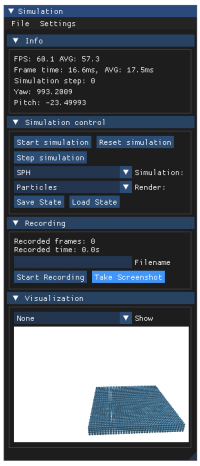
[19] https://vulkan-tutorial.com/

`Buffer` či `Image`, byly vytvořeny „`Builder`“ třídy usnadňující tvorbu daných komponent. Není tak nutné vytvářet složité konstruktory a plnit často obrovské struktury. Pro vytvoření stačí konstruktoru předat inicializovaný `Builder` s několika málo dalšími komponentami. Tento přístup má nespornou výhodu při vytváření několika podobných objektů, lze tak vytvořit jednu instanci `Builder` objektu a tu následně jen lehce upravovat před použitím v každém konkrétním objektu.

(a) **Okno s nastavením simulace**



(b) **Okno s nastavením zobrazování**



(c) **Hlavní panel**

Obrázek 4.5: **Uživatelské rozhraní.** Na obrázku 4.5a je zobrazeno okno pro nastavení simulace. Lze zde nastavovat parametry jednotlivých typů simulací. Po uložení dojde k resetování simulace a načtení nových parametrů. Při nastavování vizuálních parametrů v okně 4.5b dochází k okamžitému propsání a není nutné resetovat simulaci. Na obrázku 4.5c je následně zobrazeno hlavní okno rozhraní. Jednotlivé panely lze schovávat. Zde je uživateli poskytnuta kontrola nad celou simulací, kterou může libovolně nahrávat a spouštět a zastavovat.

4.6.1 Uživatelské rozhraní

Důležitou součástí celého jádra je instance třídy `SimulationUI`. Ta se stará o inicializaci, vykreslování, ale i zpracování událostí v rámci grafického uživatelského rozhraní. Celé rozhraní je pak vytvořeno za pomocí knihovny `pf_imgui` [20] založené na knihovně `Dear ImGui` [21]. Knihovna `pf_imgui` je stále ve vývoji a je vyvíjena studentem Fakulty informačních technologií. V tomto projektu je využívána ve verzi `v0.4.2` a je používána především kvůli zjednodušení tvorby a správy jednotlivých komponent. Některé části rozhraní jsou zobrazeny na obrázku 4.5, avšak stručně řečeno hlavní uživatelské rozhraní je obsaženo v jednom

[20] https://github.com/PetrFlajsingr/pf_imgui [21] https://github.com/ocornut/imgui

okně rozděleném do panelů, schopných skrývat svůj obsah. V těchto panelech pak uživatel může ovládat či nahrávat celou simulaci. V kontextovém menu má následně uživatel přístup k nastavení simulace.

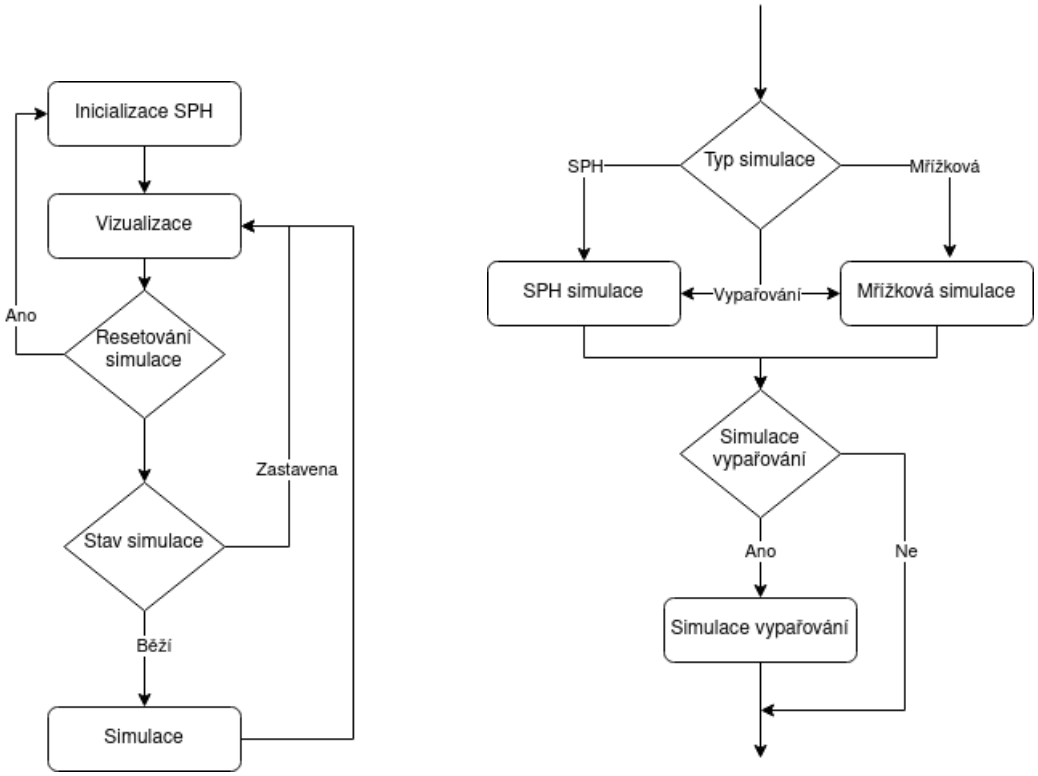
Před vytvářením samotných komponent je nutné instanci zmíněné třídy představující uživatelské rozhraní naplnit potřebnými komponentami, jako je například samotné okno či zařízení nebo instance Vulkanu. Dalším důležitým krokem je nahrát vyplněné struktury popsané v kapitole 4.5. Díky tomu je zajištěna konzistence informací pro simulaci a zobrazených údajů pro uživatele. Následným krokem je tvorba samotných komponent rozhraní, které načítají výchozí data ze zmíněných struktur. Důležitou součástí je i reaktivnost aplikace, tedy propojení akce provedené v grafickém rozhraní do samotné simulace, které je zajištěno voláním příslušných funkcí z třídy rozhraní. Máme-li například tlačítko pojmenované `buttonStartSimulation`, je volán odkaz na funkci se jménem `onButtonStartSimulationClick()`. Tyto odkazy na funkce jsou pak naplňovány tzv. „settery“. Společně se zavoláním příslušné funkce je také z třídy `SimulationUI` často předán parametr obsahující důležitá data vztahující se k dané akci. Tato data mohou představovat například vybranou barvu či výčtový typ označující vybranou položku z rolovacího menu. Pomocí volání funkcí a předávání zpráv je zajištěná komunikace mezi grafickým rozhraním a samotným jádrem aplikace.

4.7 Simulace

Simulace je hlavní částí celé aplikace. Samotné řízení simulace probíhá z již zmíněné instance třídy `VulkanCore`, nicméně sama třída neprovádí žádnou simulaci, ale pouze spouští příslušné moduly odpovídající jednotlivým typům simulace. Veškeré algoritmy jsou spouštěny na grafické kartě z důvodu urychlení a snadného mapování na paralelní architekturu. Na obrázku 4.6a lze vidět průběh jednoho kroku celé aplikace včetně vizualizace. Samotná simulace běží pouze v případě, že ji uživatel spustil, případně si vyžádal pouze jeden další krok simulace. Za zmínku stojí blok nazvaný „Inicializace SPH“. Tento blok provádí částečný výpočet částicové simulace a je nezbytný pro správné zobrazování povrchu kapaliny při metodě Marching cubes. Jelikož daná metoda spoléhá na určité vypočítané hodnoty, je nutné je předpočítat, jinak by nebyl produkován korektní výsledek při výchozím stavu celého simulátoru.

Na obrázku 4.6b je detailněji zobrazen blok samotné simulace a rozhodování, která simulace bude zrovna prováděna. Dochází-li ke kombinované simulaci, tedy k simulaci vypařování, jsou nejprve současně spuštěny obě simulace, simulace částicová (v obrázku SPH) a mřížková. Tyto simulace lze spustit současně, jelikož se jedná o nezávislé výpočty, tudíž mohou lépe zaplnit výpočetní prostor samotné grafické karty a teoreticky ušetřit výpočetní čas. Před samotným výpočtem vypařování je však nutné tyto procesy synchronizovat a počkat, až oba doběhnou. Po skončení se pokračuje do grafu 4.6a a dochází k vizualizaci výsledku.

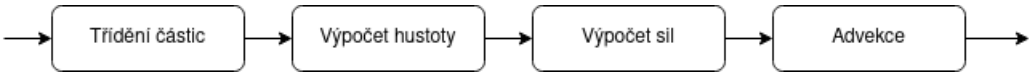
Mapování na grafickou kartu následně probíhá v závislosti na počítaném problému. Jedno vlákno odpovídá jedné částici v částicové simulaci a jedné buňce v mřížkové simulaci. U vypařování pak závisí na prováděném kroku. Při přenosu hmotnosti a teploty ve směru buňka → částice dochází k mapování vláken na částice, ve všech ostatních dochází k mapování na buňky. Velikost pracovní skupiny je následně všude stejná a jedná se o velikost 32.



(a) **Zobecněný postup při simulaci** (b) **Detail bloku Simulace z obrázku 4.6a**

Obrázek 4.6: **Diagram simulace.** Na obrázku 4.6a je zobrazen zobecněný tok programu během simulace. Jedná se o kontrolu simulace, tedy pokud uživatel vyžaduje simulaci, dochází k jejímu výpočtu, jinak je simulace pozastavena a dochází pouze k vykreslování. Na obrázku 4.6b je následně podrobněji rozkreslen blok Simulace z diagramu 4.6a. V podrobném nákresu je následně zobrazeno rozhodování o prováděném typu simulace.

4.7.1 Smoothed particle hydrodynamics



Obrázek 4.7: **Diagram SPH simulace.** Na diagramu jsou popsány jednotlivé kroky SPH simulace, tyto kroky jsou na sobě závislé, proto je každý krok rozdělen do vlastního bloku a je implementován ve vlastním shaderu. Celkově se jedná o implementaci z kapitoly 2.3.1.

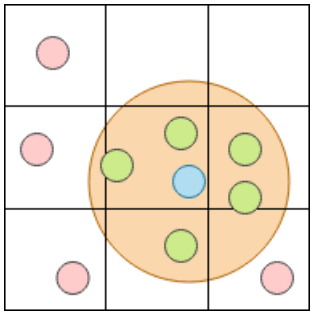
Částicová simulace vychází z algoritmu popsaném v kapitole 2.3.1 vycházejícího z publikace [22] Mickyho Kelagera z University of Copenhagen. Tento algoritmus je představován především bloky *Výpočet hustoty*, *Výpočet sil* a *Advekce* z diagramu 4.7. Jediný blok, který je dále rozdělený na více částí, je blok s výpočtem hustoty částice. Zde je kromě výpočtu hustoty částice proveden navíc výpočet středu hustoty, který vychází z článku [31]. Tento výpočet je nezbytný, jelikož velikost plochy, kterou částice zaujímá v kapalině, ovlivňuje výpočet míry vypařování. Rozdělení do těchto bloků je pak dáno především závislostí výpočtů. Většina výpočtů v bloku *Výpočet sil* vyžaduje současnou hodnotu hustoty částic,

proto je zapotřebí nejprve zmíněnou hustotu vypočítat pro všechny částice, a až poté přejít k dalším výpočtům.

Urychlení vyhledávání

Pro výpočet nových hodnot částice během simulace je z principu rovnice 2.9 nutné počítat se všemi částicemi v okolí. Nicméně ne všechny částice nutně přispívají k celkovému výsledku. Částice nacházející se ve vzdálenosti větší než je vyhlazovací poloměr h nepřispívají k výpočtu. Při nízkém počtu částic se nejedná o nijak závažný problém, avšak při vysokém počtu dochází k tomu, že při výpočtu procházíme například desetitisíce částic, ale k samotnému výpočtu přispěje pouze několik málo desítek částic.

Z tohoto důvodu se jeví jako více než vhodné využít metody schopné získat pouze částice přispívající k výpočtu hodnot. Autor publikace [22] navrhuje využití prostorového hašování, nicméně tento postup byl odmítnut ve prospěch homogenní mřížky. Hlavním důvodem byla možnost snadnějšího propojení s mřížkovou metodou simulace. Tato metoda sice neomezuje průchod pouze na částice spadající pod velikost vyhlazovacího poloměru, přesto výrazně redukuje počet částic, se kterými je nutné počítat. Princip metody je zobrazen na obrázku 4.8. Principem je rozdělení částic do mřížky, kde každá buňka má velikost rovnou vyhlazovacímu poloměru. Pokud tedy počítáme s Moorovým okolím, máme zajištěno, že částice nacházející se v tomto okolí jsou množinou částic obsahující mimo jiné všechny částice potřebné k výpočtům.

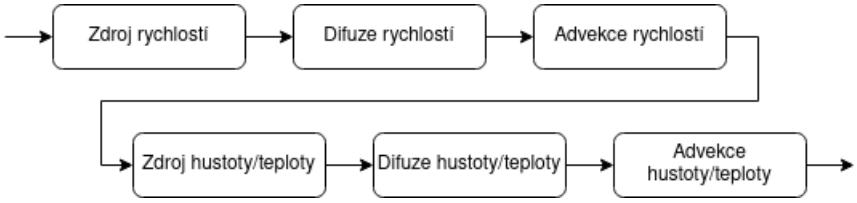


Obrázek 4.8: **Mřížková metoda pro urychlení vyhledávání.** Na obrázku je modře znázorněna částice, pro kterou jsou hodnoty počítány, zelené jsou částice potřebné pro výpočet a červené jsou částice, které budou zahrnuty do výpočtu, ale nejsou potřebné. Oranžový kruh pak znázorňuje vyhlazovací poloměr. Zobrazení ve dvou dimenzích lze samozřejmě snadno převést do trojrozměrného prostoru.

Celý průběh použitého algoritmu probíhá následovně. Nejprve je dle pozice vypočtena příslušnost částice k buňce. Tato informace je zapsána do separátního pole společně s pořadovým číslem částice v poli. Následně je využito paralelního řazení a celé takto vytvořené pole indexů částic a buněk je seřazeno dle indexu buňky. Pro řazení byl zvolen algoritmus Count sort [38] využívající paralelní verze algoritmu prefix sum [19]. Výsledkem je seřazená posloupnost, přičemž indexy částic ve stejné buňce se nacházejí za sebou. Posledním krokem je tvorba dalšího pole s indexy ukazujícími na začátek jednotlivých bloků částic nacházejících se ve stejné buňce. Takový index lze následně využít k nalezení konkrétní buňky s částicemi. Pokud je buňka prázdná, obsahuje speciální hodnotu -1 značící prázdnost. Při následném prohledávání okolí částice stačí znát pouze index buňky, který označí seznam příslušných částic nacházejících se v této buňce.

4.7.2 Mřížková simulace

Další součást obrázku 4.6b nutnou popsat je mřížková metoda simulace, která je zobrazena na obrázku 4.9. Tu lze rozdělit na dvě části skládající se následovně z několika dalších kroků. Dva hlavní bloky představují výpočty zaměřující se na rychlost a výpočty zaměřující se na hustotu s teplotou. Obě části mají téměř stejné kroky, a proto budou vysvětleny obecně, přičemž budou samozřejmě popsány případné jednotlivé odlišnosti.

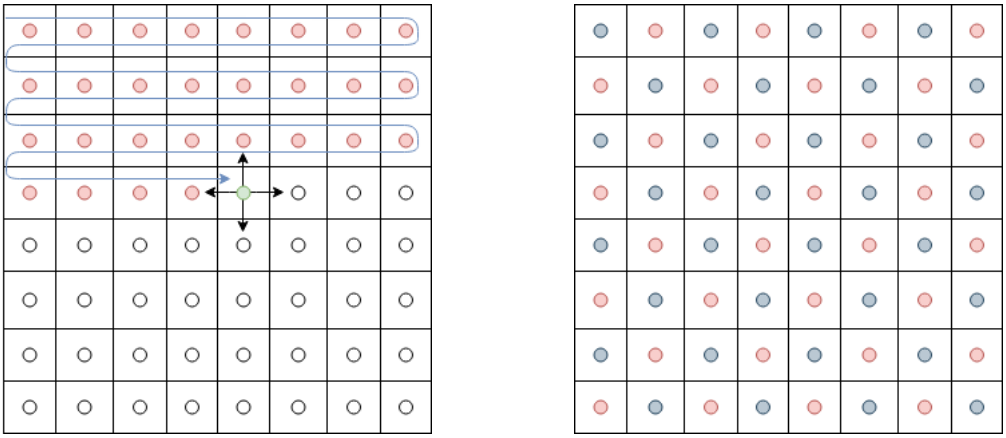


Obrázek 4.9: **Diagram mřížkové simulace.** Jednotlivé kroky mřížkové simulace. Jedná se o dva téměř totožné výpočty za sebou. Nejprve dojde k výpočtu rychlostí, přidají se případné hodnoty zdrojů, nastane difuze rychlostí a nakonec dojde k posunu rychlostí v prostoru. Následně jsou stejné kroky provedeny zároveň pro teplotu a hustotu nesených výparů. Tento postup musí být více méně zachován, jelikož v advekci hustot a teplot dochází k využití již vypočtených rychlostí v první části simulace.

Blok zdrojů je poměrně jednoduchý a dochází v něm pouze k přidávání hodnot do jednotlivých částí, jedná se tedy o určité působení zdrojů v prostoru. Do pole rychlostí se následně ještě připočítává působení hustoty a teploty podle vzorce 3.3. Poté je přistoupeno k difuzi, kdy dochází k rozptýlení hodnot do okolí. V tomto případě však může dojít k vzniku nenulové divergence, a proto je nutné aplikovat projekci a tuto divergenci odstranit, případně minimalizovat, čehož se dosahuje za pomocí Gauss-Seidlovy numerické metody, využívané i při výpočtu difuze. Posledním krokem je advekce, která je implementována pomocí tzv. backtrackingu popsaném v kapitole 2.2.1. Advekce hodnot hustoty a teploty pak využívá již vypočtených hodnot rychlosti a je tak nutné vyčkat s výpočtem na advekci rychlosti.

Problém nastává při aplikování Gauss-Seidlovy numerické metody. Principem této metody je průchod pole a postupný výpočet nových hodnot, jak je znázorněno na obrázku 4.10a, kde červeně jsou označeny již vypočtené hodnoty a černé šipky označují, ze kterých hodnot se nová hodnota počítá. Tento postup je však neproveditelný na paralelních architekturách. Z tohoto důvodu byl zvolen jiný přístup. Vhodnější metodou je například využití algoritmu Gauss-Seidel Red Black [11]. Tento přístup využívá střídavého počítání hodnot v červených a černých bodech zobrazených na obrázku 4.10b. Tím je docíleno, že jsou na sobě závislé pouze kroky střídání barev a lze tedy daný přístup využít nad paralelní architekturou. Tato metoda sice vyžaduje podle autorů článku [11] dvojnásobný počet iterací pro správnou konvergenci, nicméně pro větší počet buněk dochází k výraznému urychlení právě díky paralelním výpočtům.

V samotné aplikaci je pak invokován stále stejný výpočetní shader a pro dělení mezi fázemi výpočtu je využito specifické informace ve struktuře 4.3. Ta představuje bitové pole nesoucí mimo jiné informaci o barvě právě prováděné iterace. Díky této informaci jsou následně upraveny indexy pro přístup do pole hodnot.

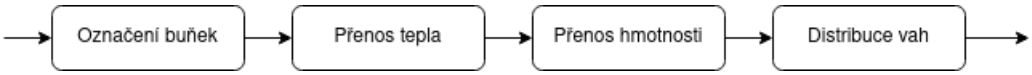


(a) **Gauss-Seidlova numerická metoda** (b) **Paralelní verze Gauss-Seidlovy Metody**

Obrázek 4.10: **Srovnání běžné a paralelní Gauss-Seidlovy numerické metody.** Na obrázku 4.10a je vyznačen výpočet pomocí běžné metody. Tato metoda počítá novou hodnotu buňky z buněk v okolí tak, jak v obrázku naznačují černé šipky. Modrá šipka značí postup celou mřížkou při výpočtu. Červeně jsou následně označeny buňky, u kterých již byl proveden výpočet. Jelikož metoda pracuje bez pomocných polí, existují v mřížce jak hodnoty nové, tak hodnoty staré. Zelená buňka označuje právě počítanou buňku. Z obrázku je zjevné, že výpočet současné hodnoty závisí jak na minulých hodnotách, ale zároveň i na hodnotách nově vypočítaných. Na obrázku 4.10b je zobrazena paralelní modifikace této metody. Výpočty se v této verzi alternují mezi černými a červenými buňkami. Takto je docíleno, že okolí právě počítané buňky není přítomno v probíhajícím výpočtu.

4.7.3 Vypařování

Poslední částí je simulace vypařování, k té však dochází pouze po kompletním výpočtu předchozích dvou simulací. Samotná simulace je pak zobrazena na diagramu 4.12. Důležitou součástí celého procesu je označení jednotlivých buněk. Každá buňka dostane dvě značky nesoucí informace o buňce. První je, zda se jedná o buňku obsahující částice. Během simulace vypařování dochází k přenosu hmotností mezi částicovou simulací a simulací mřížkovou. Označením, jestli buňce náleží nějaká část kapaliny, můžeme následně zamezit vypařování do míst obsahujících kapalinu. Druhou značkou přiřazenou buňkám v první části simulace je, zda se jedná o rozhraní mezi kapalinou a okolím. Buňka je vyhodnocena jako rozhraní, jedná-li se o buňku mající ve svém nejbližším okolí buňku jiného typu, tedy buňku



Obrázek 4.11: **Diagram simulace vypařování.** Jednotlivé kroky při simulaci vypařování. Důležitým krokem je označení buněk. Zde dochází k přidání značek k jednotlivým buňkám. Tyto značky jsou následně využity v dalších krocích, kde v závislosti na značce například nemusí vůbec nastat výpočet vypařování. Posledním blokem je distribuce vah. Zde dochází k zániku částic a vyrovnání vah, pokud jsou splněny příslušné podmínky. Tento blok zajišťuje určitým způsobem vyrovnávání velikosti částic, což pomáhá stabilitě simulace a zabraňuje interakci částic různých velikostí.

prázdnou či obsahující kapalinu. Jako okolí pro vyhodnocení rozhraní bylo zvoleno trojrozměrné Von-Neumanovo okolí, kde jeho dvojrozměrná verze je zobrazena na obrázku 2.9. Tímto přístupem vzniká označení rozhraní z obou stran povrchu kapaliny. Toto označení je důležité z hlediska přenosu tepla mezi kapalinou a okolím, jelikož přenos tepla nastává pouze na povrchu kapaliny.

Samotný blok přenosu tepla se pak v aplikaci rozděluje na přenos tepla z kapaliny do okolí a z okolí do kapaliny. Jelikož je implementováno pouze vypařování, přenos hmotnosti funguje pouze jednosměrně a využívá váhových faktorů zmíněných na konci kapitoly 3.2. Na konci simulace pak dochází k vyrovnávání zmíněných faktorů. Existuje-li částice s váhou $w < 0,5$, dochází k jejímu zániku a distribuci její zbylé váhy do okolních částic. Využití těchto vah spolu nese nutnost začlenit tyto váhy do výpočtů samotné částicové simulace, ale také do ostatních částí simulace, kde například částice s váhou $w = 0$, tedy zaniklé částice, nejsou brány v potaz při výpočtu výsledku. Příkladem může být například zmíněné značení buněk.

4.8 Vizualizace

Důležitou součástí celé aplikace je také vizualizace výsledku simulace, bez které by výsledkem byla jen změť pozic, rychlostí a jiných číselných hodnot. Vizualizaci je nutné provádět jak pro částicovou simulaci, tak pro simulaci mřížkovou, níže jsou pak popsány použité metody pro zobrazování jednotlivých simulací.

4.8.1 Smoothed particle hydrodynamics

Ve vytvářené aplikaci byly použity metody přímé vizualizace částic a extrakce povrchu pomocí Marching cubes. Mezi těmito metodami lze následně libovolně přepínat. Důležitou součástí je také možnost vizualizace hustoty jednotlivých částic pomocí barevného rozložení.

Vizualizace částic

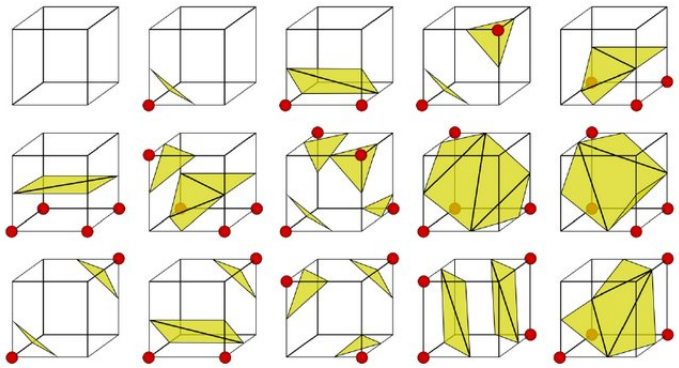
Vizualizace částic je tou nejprimitivnější metodou vizualizace schopnou vykreslit jednotlivé částice. Jedná se o poměrně nenáročnou metodu, díky níž lze zobrazovat jednotlivé výpočetní částice v kapalině. Hlavním principem je pak technika instancovaného vykreslování. V paměti grafické karty je uložen jeden model částice a ten je následně vykreslován několikrát za sebou, aby počet vykreslování odpovídal počtu částic. Na každou jednotlivou instanci modelu je následně aplikována translace a škálování, aby daná instance odpovídala pozici konkrétní částice a její váze w.

Tento typ vykreslování byl implementován již na začátku celého vývoje a jeho nespornou výhodou je možnost sledovat chování jednotlivých částic. Zvolený model koule přináší výhodu totální symetrie a odpadá tedy nutnost určitým způsobem rotovat částicí, aby působila dojmem reálného pohybu, jako by tomu bylo například u krychle.

Marching cubes

Metoda Marching cubes [26] je metoda extrakce povrchu za pomoci předem definované funkce, tzv. signed distance function. Tato funkce dokáže říci, na jaké straně objektu se daná pozice nachází a jaká je vzdálenost k jeho povrchu. S její pomocí lze navzorkovat daný prostor rozdělený do mřížky a v každém vrcholu buňky mít hodnotu o vzdálenosti k povrchu. O buňkách neležících čistě vně či uvnitř objektu lze následně tvrdit, že jimi prochází povrch

daného modelu. V průběhu let pak bylo vytvořeno několik modifikací vyhledávacích tabulek určujících jaké hrany buňky propojit, aby výsledkem byl „vodotěsný“ model aproximující danou funkci.



Obrázek 4.12: **Vyhledávací tabulka Marching cubes.** Vyhledávací tabulka znázorňující jednotlivé možnosti, jak sestavit polygony v dané buňce. Červeně jsou označeny body ležící uvnitř modelu. Na obrázku je následně znázorněno pouze 15 případů, zbylé případy jsou totiž symetrické a lze je získat transformací představených možností.
Zdroj: Isovox: A Brick-Octree Approach to Indirect Visualization [23]

V samotné aplikaci je algoritmus řešen pomocí výpočetních a geometrických shaderů. Výchozí velikost mřížky je stejná jako velikost mřížky pro usnadnění vyhledávání popsané v kapitole 4.7.1. Výchozí mřížka je však velmi hrubá a produkuje poměrně hrubý model. Z toho důvodu je v aplikaci možnost zvýšit detail výsledného modelu pomocí rozdělení každé buňky na několik dalších, čímž je docíleno jemnějšího vzorkování při výpočtu a následně detailnějšího modelu. Pro vzorkování je pak využita funkce výpočtu barevného pole 2.27, přičemž jako práh rozdílu mezi kapalinou a okolím byla zvolena hodnota 0,5. Samotné vyhledávací tabulky byly převzaty z projektu Terrain generation [22]. [4] [1]

Při samotném vykreslování pak dochází opět k instancovanému vykreslování, nicméně nyní se jedná pouze o body představující jednotlivé buňky. Do vykreslovacího toku je zařazen geometrický shader, který dle navzorkované mřížky a kompletní vyhledávací tabulky produkuje polygony představující povrch kapaliny.

4.8.2 Mřížková simulace

Prvním pokusem pro mřížkovou simulaci především pro testování bylo obdobné vykreslování jako u vizualizace částic. V tomto případě však došlo k instancovanému vykreslování krychlí černé barvy. Každá krychle měla nastavenou průhlednost v závislosti na množství obsažené v dané buňce. Různě průhledné buňky byly následně míchány mezi sebou a vznikala iluze kouře. Bohužel tato iluze nebyla dostatečně věrná, a proto bylo přistoupeno k jiné metodě.

Výsledná použitá metoda je pak modifikací na vykreslování zmíněné ve článku Visual Simulation of Smoke [14]. Každá buňka obsahující nějaké množství látky je vykreslována jako čtverec zarovnaný s osou, která je nejvíce paralelní s osou pozorovatele. Následně jsou čtverce vykresleny, nicméně každý vrchol je brán jako jiná buňka, a tak dochází k interpolaci průhledností mezi jednotlivými vrcholy čtverce. Jedná se o poměrně jednoduchou techniku, která ale produkuje vcelku působivé výsledky.

[22] https://github.com/PetrFlajsingr/TerrainGeneration

Kapitola 5

Testování

Chod celého simulátoru byl testován především během vývoje. Po přidání nové funkčnosti, jako je například nový způsob vizualizace, případně nový modul pro simulaci, byl otestován chod nejen současného modulu, ale i stávajících součástí systému. Po vytvoření nového modulu nebo refaktorizaci kódu totiž často docházelo k porušení synchronizace mezi jednotlivými invokacemi výpočtů. Výsledná aplikace se tak často stala částečně, ale někdy i úplně nefunkční. V těchto případech však byly velmi nápomocné validační vrstvy Vulkanu. Tyto vrstvy hlásily nejen kritické chyby zapřičiňující zamrzání či pád aplikace, ale také nesprávné používání samotného API a tedy možné snížení výkonu.

Testování a ladění bylo často velmi komplikované. Ladit stranu aplikace běžící na klasickém procesoru je vcelku běžná záležitost krokování a zkoumání hodnot proměnných. Avšak ladění výpočtů běžících na grafické kartě je o poznání složitější. Programátor nemá možnost kdykoliv nahlédnout do obsahu paměti a nemá možnost ani spuštěný výpočet v libovolný okamžik zastavit. Proto bylo implementováno několik funkcí, které měly tento proces usnadnit. Jedním z hlavních pomocníků byla funkce čtení paměti z grafické karty. Často bylo postupováno tak, že byl program zastaven ihned po dokončení výpočtu a přenesení dat z grafické karty do paměti počítače. Zároveň byly v záznamech částic vytvářeny pomocné proměnné, do kterých byly ukládány mezivýsledky. Celý proces ladění a hledaní chyb usnadnilo objevení rozšíření `GL_EXT_debug_printf`. Toto rozšíření po správném nastavení umožňuje diagnostický výpis přímo z grafické karty do validačních vrstev. Zmíněného rozšíření bylo následně hojně využíváno, a společně s předešlou možností nahlédnutí do paměti poskytovaly tyto nástroje dostatečný způsob pro ladění celé aplikace.

Každá simulace byla otestována s různými velikostmi simulačních prostorů. Pro ladění a hledání chyb bylo často využíváno malého počtu částic, případně buněk. To poskytlo snadnější orientaci ve výpisech a umožňovalo i ruční nákresy či výpočty. Po odladění na malém prostoru byl algoritmus také otestován na větších objemech a případně laděn dál. Jelikož velké množství simulace se odehrává nad homogenní mřížkou, byl velice nápomocný čtverečkovaný papír, kde bylo možné jednotlivé situace rozkreslit a zapsat si například jednotlivé indexy a přemýšlet nad řešením problému.

Smoothed particle hydrodynamics

Největším problémem této metody bylo nalezení správných parametrů simulace. Ačkoliv byl algoritmus správný, se špatnou kombinací parametrů, jako je například dlouhý krok simulace a vysoká viskozita, dojde k nestabilitě simulace. Dalším velkým problémem bylo výchozí postavení částic. Pokud byly částice natěsnány velmi blízko u sebe, došlo v prv-

ním kroku simulace k výpočtu velmi vysoké hustoty jednotlivých částic a s tím spojeným vysokým tlakem. Za pomoci vysokého tlaku byla následně vypočtena rychlost tak, aby byl neprodleně redukován tlak a hustota na hodnoty odpovídající dané kapalině. Z toho důvodu byla vypočítaná rychlost velmi vysoká a reálně docházelo k „explozi“ dané kapaliny. Při opačném stavu, kdy byly částice velmi daleko od sebe, docházelo k vyrovnání hodnot tak, že celý model „implodoval“ a opět se simulace stala nestabilní.

Mřížková simulace

Testovat mřížkovou simulaci bylo velmi obtížné z důvodu mnoha kroků simulace. Najít krok, kde se nachází chyba, bylo obtížné, stejně tak jako interpretovat správnost výstupů ve formě vektorů rychlosti a hodnot teploty a hustoty. Nápomocná však byla již hotová funkční implementace [1]. S touto implementací byly následně srovnávány výsledky jednotlivých kroků a pokud byly v rámci mezí, bylo přistoupeno k testování dalšího kroku.

Další důležitou částí, která dělala problém, byl výpočet správnosti používaných indexů, především pak při paralelní Gauss-Seidlově metodě, kde je nutné indexovat pouze některé buňky. Zde bylo velmi nápomocné testovat a ladit danou část nad malým počtem buněk, rozepsat si jednotlivé indexy a zjišťovat nesrovnalosti ve výpočtech.

5.1 Výsledek

Výsledná aplikace plní své vytyčené cíle. Je schopná simulovat kapalinu i vypařování a poskytuje uživateli možnost ovlivňovat průběh simulace. Výslednou podobu celého simulátoru lze vidět na obrázcích 5.1, 5.2, 5.3 a 5.4. Důležitým faktorem je samozřejmě i rychlost výpočtu simulace. Celá implementace probíhala na systému s procesorem AMD Ryzen 5 5600X a grafické kartě Nvidia RTX 3070. Ačkoliv systém poskytuje poměrně dost výkonu, samotná aplikace bohužel neběží moc rychle. V tabulce 5.1 je zobrazeno několik scénářů, jejich důležité parametry a rychlost výpočtu.

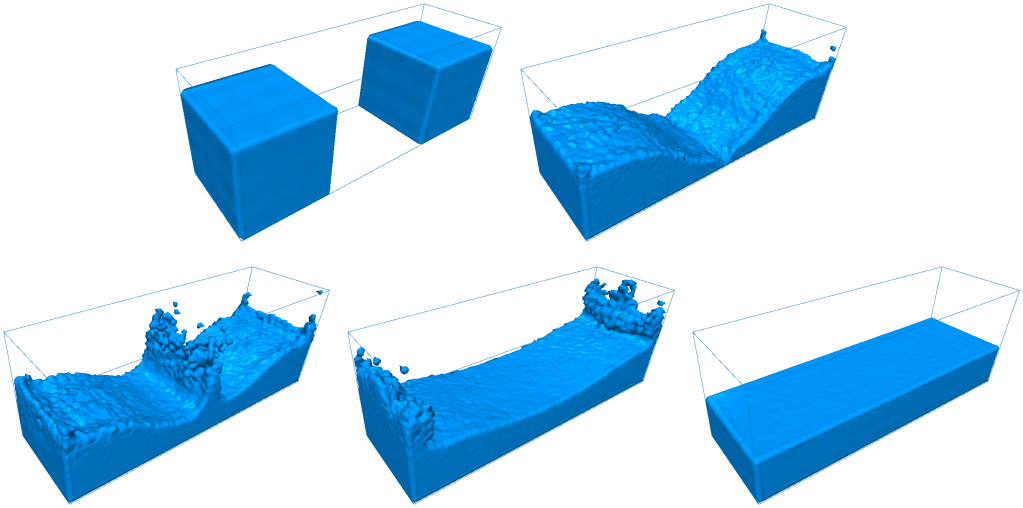
Scénář	Počet částic	Velikost mřížky	SPH simulace	Mřížková simulace
Kapalina v nádobě	27000	-	3,8 ms	-
Kolize	55800	-	6,0 ms	-
Vypařování	5476	20^3	2,6 ms	16,8 ms

Tabulka 5.1: **Srovnání časů různých scénářů.** V tabulce je možné porovnat tři různé scénáře a jejich časy na výpočet jednoho snímku. Krok na simulaci byl nastaven na $dt = 0,001$ a celkově byla simulace nastavena tak, aby byla co nejvíce nestlačitelná.

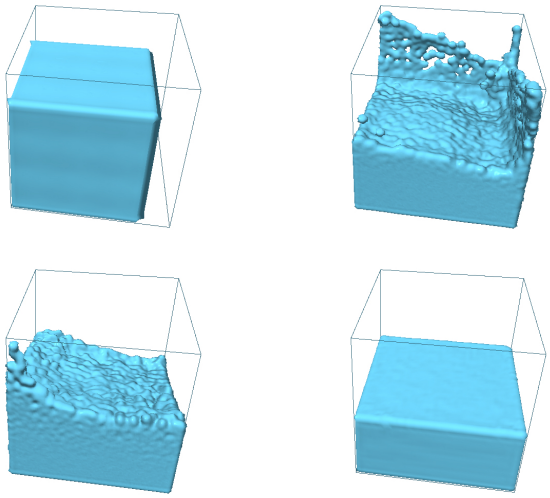
Délka výpočtu samozřejmě závisí na nastavených parametrech. Pro kvalitní téměř nestlačitelnou simulaci je důležité využít malého simulačního kroku a vysoké hodnoty plynné tuhosti. Osvědčilo se využití hodnot $dt = 0,001$ a $k = 100$. Pokud je snížena hodnota tuhosti plynu $k = 10$ a naopak zvýšen simulační krok $dt = 0,01$, dojde v některých scénářích k simulaci v téměř reálném čase. Obětí v tomto případě je však nestlačitelnost kapaliny, kdy dochází k celkem silnému stlačování kapaliny při působení sil. V klidovém stavu však kapalina nevykazuje téměř žádné abnormality. Při simulaci vypařovaní je náročná mřížková metoda. Jak je patrné z kapitoly 4.7.2 metoda má mnoho kroků a především během svého výpočtu využívá několikrát Gauss-Seidlovy numerické metody, kterou je nutné spouštět

[1] https://github.com/BlainMaguire/3dfluid

v několika iteracích. Posledním problémem je vykreslování za pomocí Marching cubes. Zde záleží výkonnost na detailu mřížky, ale již při trojnásobném dělení dochází k výraznému zpomalení aplikace. Největším problémem je výpočet hodnot v mřížce, jelikož v každém bodě mřížky je nutné vypočítat hodnotu barevného pole. Výpočet barevného pole je výpočtem SPH simulace a k dosažení výsledku jsou tedy využívány okolní částice, což má za následek vysoký počet výpočtů a procházení paměti. Částečné urychlení by mohlo nastat při lepším využití paměti, nicméně vhodnější by byl nejspíš algoritmus z principu produkující výsledek jak rychleji tak zároveň kvalitněji.

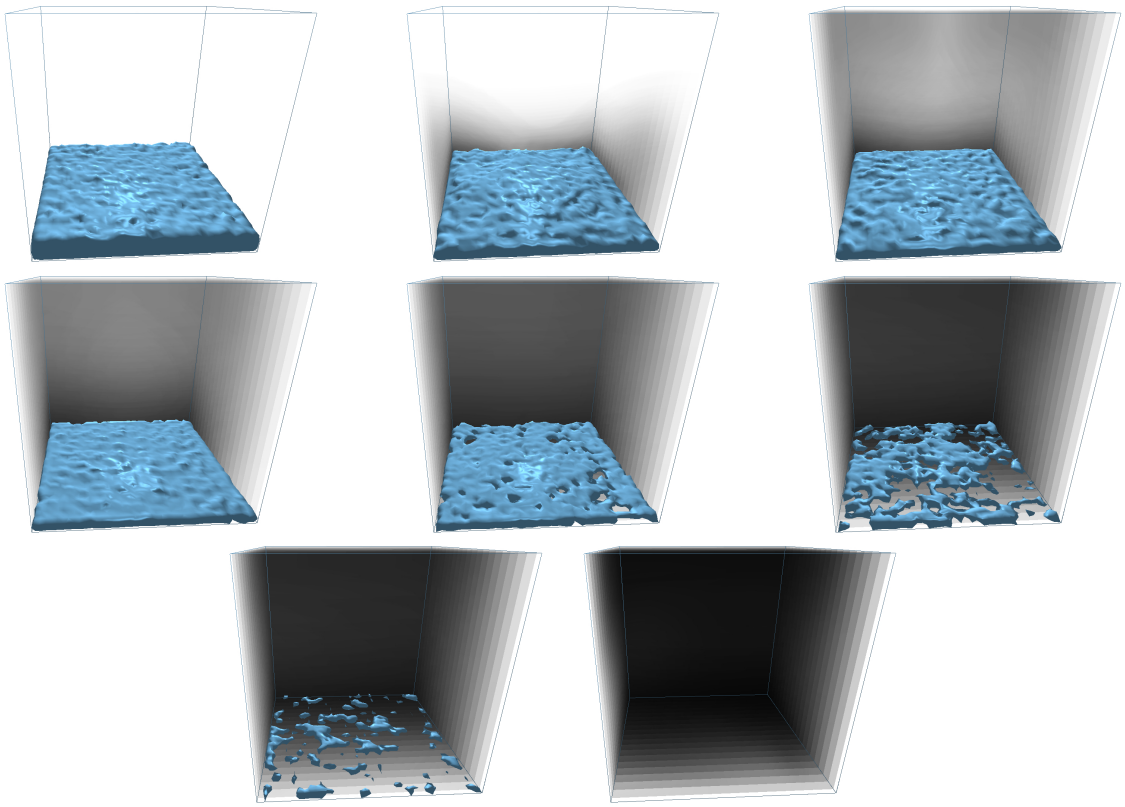


Obrázek 5.1: **Kolize dvou kapalin.**



Obrázek 5.2: **Kapalina v nádobě.**

Obrázek 5.3: **Srovnání vykreslené hladiny a zobrazených částic.**



Obrázek 5.4: **Simulace vypařování.**

Kapitola 6

Závěr

Cílem celé práce bylo vytvořit aplikaci schopnou simulovat chování kapaliny a tím se seznámit s principy simulování kapalin. K dosažení tohoto cíle bylo nutné nejdříve nastudovat jednotlivé algoritmy. Toto studium se pak sestávalo nejen ze studia algoritmů, které byly následně implementovány, ale také jiných metod, pomocí nichž lze simulovat chování kapalin.

Následný návrh se dále zabýval nejen samotnou funkčností aplikace, ale bylo také stanoveno, že projekt bude implementovat dva různé přístupy k simulaci, tedy metodu Smoothed Particle Hydrodynamics a Eulerovu mřížkovou metodu. V souvislosti s tím byl vybrán a nastudován článek [20] zaměřující se na vypařování a kondenzaci propojením těchto dvou metod. Následně zde byly představeny principy a rovnice nutné k jeho implementaci. Zároveň bylo řečeno, že nedojde k implementaci kondenzace, nýbrž bude implementováno pouze vypařování kapalin. Další důležitou součástí návrhu byl výběr grafického API, využitého při následné implementaci. Bylo vybíráno z několika kandidátů, přičemž každý byl krátce rozebrán, byly zmíněny jeho výhody a nevýhody a na konci byl učiněn finální výběr. Jako výsledné API byl zvolen Vulkan, především pro jeho rostoucí popularitu a kontrolu, kterou programátorovi nabízí. Poslední částí návrhu se stal návrh grafického rozhraní. Zde byly navrhnuty ovládací prvky pro celou aplikaci, které uživateli usnadní používání celé aplikace.

Pro implementaci bylo využito zmíněného API Vulkan, pomocí kterého bylo dosaženo plné simulace na grafické kartě a využito tak paralelního potenciálu celého problému. Během implementace bylo nutné řešit řadu problémů, především nutnost studia některých paralelních algoritmů, jelikož některé články poskytovaly algoritmy nevhodné pro paralelní architektury.

Výsledná aplikace je funkční a produkuje poměrně dobré výsledky, nicméně jsou zde oblasti, které je možné dále rozpracovat a dosáhnout tak lepších výsledků. Jedná se celkově o poměrně komplexní téma, a proto při implementaci byly některé postupy zjednodušeny. Největším zjednodušením celého projektu bylo rozhodnutí implementovat pouze simulaci vypařování a kondenzaci vynechat. Dalším problémem byla částečná stlačitelnost kapaliny při částicové simulaci. V aplikaci byla použita metoda nazvaná WCSPH (Weakly Compressible Smoothed Particles Hydrodynamics), která, jak je z názvu poznat, umožňuje kapalinu lehce stlačit. Míru stlačitelnosti lze částečně ovlivnit pomocí velikosti kroku simulace a hodnoty tuhosti plynu, což ale vytváří vyšší výpočetní náročnost.

Je důležité také zhodnotit samotný výkon celé aplikace. V prvopočátcích vývoje, kdy nebylo použito urychlení výpočtu částicové simulace, začínala mít celá aplikace značné problémy již při tisícovce částic. Po využití mřížkové metody došlo k výraznému urychlení a simulátor zvládne vcelku rozumně simulovat desítky tisíc částic, vše však velice závisí na

velikosti zvoleného kroku simulace a zvolené tuhosti plynu. Eulerova mřížková simulace je pak mnohem pomalejší, především z důvodu mnoha kroků simulace a využití iteračních metod, nutných invokovat výpočty několikrát v cyklu.

Pro budoucí rozšíření samotné aplikace se nabízí celá řada možností. První je samozřejmě implementace zmíněné vynechané kondenzace. Další možností je implementovat pokročilejší způsob částicové simulace, jako je například ISPH (Incompressible Smoothed Particles Hydrodynamics) nebo PCISPH (Predictive–corrective Incompressible Smoothed Particles Hydrodynamics), které odstraňují stlačitelnost kapaliny, případně zlepšují celkovou stabilitu simulace a dovolují delší simulační krok. Dalším krokem by pak také mohla být optimalizace algoritmů pro zrychlení chodu všech simulací. Vylepšení by si zasloužila i prezentační stránka aplikace. Bylo by zajímavé implementovat i rozdílné metody zobrazovaní, jako například Surface Splatting [42], či využít existence mřížky a implementovat volume raycasting, případně rovnou raytracing [24]. Rovněž zajímavou funkčností by mohlo být zakomponování pevných těles a jejich fyziky. V aplikaci by následně bylo možné vidět pevná tělesa, o která by se kapalina tříštila, případně by je kapalina unášela na hladině.

Literatura

[1] *Chapter 1. Generating Complex Procedural Terrains Using the GPU* [online]. [cit. 2021-05-10]. Dostupné z: https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-1-generating-complex-procedural-terrains-using-gpu.

[2] *Definition of DirectX* [online]. [cit. 2021-05-10]. Dostupné z: https://www.pcmag.com/encyclopedia/term/directx.

[3] *OpenGL Wiki* [online]. [cit. 2021-05-10]. Dostupné z: https://www.khronos.org/opengl/wiki/.

[4] *Surface rendering of Fluids using Smoothed Particle Hydrodynamics NTU CSIE Rendering 2011/2012, Final Project by Dominik Seifert (best viewed in Chrome)* [online]. [cit. 2021-04-15]. Dostupné z: https://www.csie.ntu.edu.tw/~b97122/_archive/0302/rendering/report.html.

[5] *WebGL Wiki* [online]. [cit. 2021-05-10]. Dostupné z: https://www.khronos.org/webgl/wiki.

[6] *GPU Gems* [online]. Nvidia, Sep 2007 [cit. 2021-01-13]. Dostupné z: https://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch38.html.

[7] *Fluid Simulation (with WebGL demo)* [online]. Jamie Wong, Aug 2016 [cit. 2021-01-13]. Dostupné z: http://jamie-wong.com/2016/08/05/webgl-fluid-simulation/.

[8] *What Are the Navier-Stokes Equations?: SimScale Numerics* [online]. Oct 2020 [cit. 2021-01-14]. Dostupné z: https://www.simscale.com/docs/simwiki/numerics-background/what-are-the-navier-stokes-equations/.

[9] *CUDA Zone* [online]. May 2021 [cit. 2021-05-10]. Dostupné z: https://developer.nvidia.com/cuda-zone.

[10] 10, S. a Ebersole, M. *What Is CUDA: NVIDIA Official Blog* [online]. May 2018 [cit. 2021-05-10]. Dostupné z: https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/.

[11] Amador, G. a Gomes, A. Linear Solvers for Stable Fluids: GPU vs CPU. In:. 2012.

[12] Cleary, P. W. a Monaghan, J. J. Conduction Modelling Using Smoothed Particle Hydrodynamics. *Journal of Computational Physics*. 1999, sv. 148, č. 1, s. 227 – 264. DOI: https://doi.org/10.1006/jcph.1998.6118. ISSN 0021-9991. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0021999198961186.

[13] Desbrun, M. a Gascuel, M.-P. Smoothed Particles: A New Paradigm for Animating Highly Deformable Bodies. In: *Proceedings of the Eurographics Workshop on Computer Animation and Simulation '96*. Berlin, Heidelberg: Springer-Verlag, 1996, s. 61–76. ISBN 3211828850.

[14] Fedkiw, R., Stam, J. a Jensen, H. W. Visual Simulation of Smoke. In:. New York, NY, USA: Association for Computing Machinery, 2001, s. 15–22. SIGGRAPH '01. DOI: 10.1145/383259.383260. ISBN 158113374X. Dostupné z: https://doi.org/10.1145/383259.383260.

[15] Gingold, R. A. a Monaghan, J. J. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*. Prosinec 1977, sv. 181, č. 3, s. 375–389. DOI: 10.1093/mnras/181.3.375. ISSN 0035-8711. Dostupné z: https://doi.org/10.1093/mnras/181.3.375.

[16] GrantMeStrength. *DirectX graphics and gaming - Win32 apps* [online]. [cit. 2021-05-10]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/directx.

[17] Group, K. *The Industry's Foundation for High Performance Graphics* [online]. [cit. 2021-05-10]. Dostupné z: https://www.opengl.org//.

[18] Group, T. K. V. W. [online]. [cit. 2021-05-10]. Dostupné z: https://www.khronos.org/registry/vulkan/specs/1.2/html/.

[19] Harris, M., Sengupta, S. a Owens, J. Parallel prefix sum (scan) with CUDA. In:. Srpen 2007, sv. 39, s. 851–.

[20] Hochstetter, H. a Kolb, A. Evaporation and condensation of SPH-based fluids. In:. červenec 2017, s. 1–9. DOI: 10.1145/3099564.3099580. ISBN 978-1-4503-5091-4.

[21] Ihmsen, M., Orthmann, J., Solenthaler, B., Kolb, A. a Teschner, M. SPH Fluids in Computer Graphics. In: Lefebvre, S. a Spagnuolo, M., ed. *Eurographics 2014 - State of the Art Reports*. The Eurographics Association, 2014. DOI: 10.2312/egst.20141034. ISSN 1017-4656.

[22] Kelager, M. Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics. *Camb. Monogr. Mech.* Leden 2006, sv. 77.

[23] Laprairie, M., Ca, L. a Hamilton, H. Isovox: A Brick-Octree Approach to Indirect Visualization. Květen 2021.

[24] Levoy, M. Efficient Ray Tracing of Volume Data. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. červenec 1990, sv. 9, č. 3, s. 245–261. DOI: 10.1145/78964.78965. ISSN 0730-0301. Dostupné z: https://doi.org/10.1145/78964.78965.

[25] Liu, M. a Liu, G. Smoothed Particle Hydrodynamics (SPH): an Overview and Recent Developments. *Archives of Computational Methods in Engineering*. Březen 2010, sv. 17, s. 25–76. DOI: 10.1007/s11831-010-9040-7.

[26] Lorensen, W. E. a Cline, H. E. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for

Computing Machinery, 1987, s. 163–169. SIGGRAPH '87. DOI: 10.1145/37401.37422. ISBN 0897912276. Dostupné z: https://doi.org/10.1145/37401.37422.

[27] Lucy, L. B. A numerical approach to the testing of the fission hypothesis. . prosinec 1977, sv. 82, s. 1013–1024. DOI: 10.1086/112164.

[28] MEDVECKÝ HERETIK, J. *Simulace vody v herním prostředí [online].* 2018 [cit. 2021-01-07]. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno. Vedoucí práce Chmelík, J. Dostupné z: https://is.muni.cz/th/mfkg2/.

[29] Monaghan, J. J. Smoothed particle hydrodynamics. . leden 1992, sv. 30, s. 543–574. DOI: 10.1146/annurev.aa.30.090192.002551.

[30] Müller, M., Charypar, D. a Gross, M. Particle-Based Fluid Simulation for Interactive Applications. In:. červenec 2003, sv. 2003, s. 154–159. ISBN 1581136595.

[31] Orthmann, J., Hochstetter, H., Bader, J., Bayraktar, S. a Kolb, A. Consistent surface model for SPH-based fluid transport. In:. červenec 2013, s. 95–103. DOI: 10.1145/2485895.2485902.

[32] Overvoorde, A. *Introduction* [online]. [cit. 2021-05-10]. Dostupné z: https://vulkan-tutorial.com/.

[33] Parrish, K. *DirectX 12: what is it, and why it matters to PC gamers* [online]. TechRadar, Apr 2016 [cit. 2021-05-10]. Dostupné z: https://www.techradar.com/news/gaming/directx-12-what-is-it-and-why-it-matters-to-pc-gamers-1318636.

[34] Singh, P. *Learning Vulkan* [online]. Packt Publishing, 2016. ISBN 9781786460844. Dostupné z: https://books.google.cz/books?id=eczcDgAAQBAJ.

[35] Smith, C. C., Löf, G. a Jones, R. Measurement and analysis of evaporation from an inactive outdoor swimming pool. *Solar Energy.* 1994, sv. 53, č. 1, s. 3 – 7. DOI: https://doi.org/10.1016/S0038-092X(94)90597-5. ISSN 0038-092X. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0038092X94905975.

[36] Stam, J. Stable Fluids. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques.* USA: ACM Press/Addison-Wesley Publishing Co., 1999, s. 121–128. SIGGRAPH '99. DOI: 10.1145/311535.311548. ISBN 0201485605. Dostupné z: https://doi.org/10.1145/311535.311548.

[37] Stejskal, J. *Proudění magnetické kapaliny s aplikací Binghamova modelu.* Vysoké učení technické v Brně. Fakulta strojního inženýrství, 2013.

[38] Sun, W. a Ma, Z. Count Sort for GPU Computing. In: *2009 15th International Conference on Parallel and Distributed Systems.* 2009, s. 919–924. DOI: 10.1109/ICPADS.2009.30.

[39] Toml Lang. *Toml-lang/toml* [online]. [cit. 2021-04-20]. Dostupné z: https://github.com/toml-lang/toml.

[40] Trevett, N., Richards, A., Butler, M., McVeigh, J., Bhat, A. et al. *OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems* [online]. Jul 2013 [cit. 2021-05-10]. Dostupné z: https://www.khronos.org/opencl/.

[41] Yau, M. a Rogers, R. *A Short Course in Cloud Physics*. Elsevier Science, 1996. ISBN 9780080570945. Dostupné z: https://books.google.cz/books?id=ClKbCgAAQBAJ.

[42] Zwicker, M., Pfister, H., Baar, J. a Gross, M. Surface Splatting. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*. Srpen 2001, sv. 2001. DOI: 10.1145/383259.383300.

Příloha A

Obsah přiloženého paměťového média

- `README.TXT` - Obsahuje informace o aplikaci, jak ji přeložit a používat
- `doc/` - Složka obsahující dokumentaci
  - `thesis/` - Text a zdrojové soubory této práce
  - `Thesis.pdf` - Vygenerovaná dokumentace
- `src/` - Zdrojové soubory
- `bin/` - Složka se spustitelným souborem pro systém Linux
- `video/` - Složka s prezentačním videem

Příloha B

Konfigurační soubor

```
[App]
DEBUG = "bool"
pitch = "float"
yaw = "float"
cameraPos = ["float", "float", "float"]
lightPos = ["float", "float", "float"]
lightColor = ["float", "float", "float"]

[App.MarchingCubes]
 threshold = "float"
 detail = "int"

[App.simulationSPH]
gridSize = ["int", "int", "int"]
gasStiffness = "float"
heatCapacity = "float"
timeStep = "float"
particleModel = "string"
heatConductivity = "float"
gridOrigin = ["float", "float", "float"]
fluidDensity = "float"
viscosityCoefficient = "float"
temperature = "float"
fluidVolume = "float"

[[App.simulationSPH.Model]]
particleModelOrigin = ["float", "float", "float"]
particleModelSize = ["int", "int", "int"]

[App.simulationSPH.datafiles]
particles = "string" -optional
```

```
[App.Evaporation]
coefficientA = "float"
coefficientB = "float"

[App.simulationGridFluid]
cellModel = "string"
specificGasConstant = "float"
heatCapacity = "float"
buoyancyBeta = "float"
buoyancyAlpha = "float"
heatConductivity = "float"
diffusionCoefficient = "float"
ambientTemperature = "float"

[App.simulationGridFluid.datafiles]
valuesSrc = "string" -optional
values = "string" -optional
velocitySrc = "string" -optional
velocity = "string" -optional

[Vulkan]
pathToShaders = "string"

[Vulkan.window]
name = "string"
width = "int"
height = "int"
```